TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER'S THESIS

Code Structure Visualization

by
G.L.P.M. Lommerse

Supervisor:

Dr. Ir. A.C. Telea (TUE)

Eindhoven, August 2005

Abstract

This thesis describes a tool that helps programmers to get insight in software projects and also helps them to manage projects. The tool, called Code Structure Visualizer (CSV), visualizes the structure of source code by augmenting a textual representation of source code with a graphical representation of the structure. The structure is represented by drawing shaded cushions for each syntactic construct on top of the textual layout. The tool also provides means of interactively navigating and querying source code. The information used in visualizing the structure of source code is extracted from the code using a specially built fact extractor. The design of this fact extractor is also described in this thesis.

Contents

Chapter 1

Introduction

1.1 Problem statement

A common method of measuring the size of a software project is by counting the lines of code (LOC). Many of today's software projects are large with respect to this number. Small software projects contain thousands of lines of code, medium sized projects can easily contain over hundreds of thousands LOC and large-scale industrial software projects contain millions of lines of code.

From a programmer's point-of-view it is important both to get insight in source code and to manage source code. Many of today's source code editors provide limited ways of giving insight in the structure of source code. A common visual enhancement in code editors is lexical highlighting. This gives insight in the structure of source code on a lexical level but it is does not give a clear understanding of the higher level structures such as syntactic structures.

We propose a visualization tool, called Code Structure Visualizer (CSV), to help programmers manage and navigate large amounts of source code.

1.2 Objectives

We set out a number of objectives for CSV:

- provide a visualization that clearly depicts the structure of the source code displayed

  In order to get insight in the source code it is imperative to get insight in the structure of the source code. The source code can be enriched with a graphical representation of the structure without affecting the layout of the text. Keeping the original textual layout as a basis for the visualization is important as programmers are already familiar with the textual layout. The graphical representation must facilitate getting insight into the structure of the code without the need to actually read the code itself.

  The visualization tool builds on top of the source code and it's layout as this is what programmers are familiar to. The tool adds graphical structure information to the source code to facilitate understanding the structure of the code.

- provide means of displaying large quantities of code on a single display

  Software projects usually consist of a set of source files. To get insight in the structure of these files it must be possible to display multiple files on a single screen.

- provide means of interactively navigating source code

  Navigation is an important aid in the process of understanding the structure of source code. It must be possible to navigate on a textual, syntactic and semantic level.

- provide an extendible query system that supports interactively querying source code

  Querying source code allows for analysis of code on a syntactic and semantic level. An extendible query system allows users to extend the query system with user specific queries. This allows the user to analyze the code in a way that is not possible with the current set of available queries. Examples of queries are: "where is this identifier declared?" or "what is the base class of this class?".

1.3 Overview

Chapter 2 gives a short overview of available fact extractors. It lists specific requirements we have for a fact extractor that can be used in combination with the Code Structure Visualizer (CSV). Advantages and disadvantages of various fact extractors are given with respect to our specific requirements.

Chapter 3 describes the design of a fact extractor constructed specifically for the code structure visualization tool. The fact extractor is based on the GNU Compiler Collection (GCC) C++ compiler front-end. A detailed description is given how this compiler is modified in order to meet our specific fact extraction requirements.

The process of visualizing the facts is described in chapter 4. The chapter starts with describing the methods used in visualizing the facts and how these are implemented. Next it describes the layout of the visualized information. Following is a section on the visibility of the various graphical representations. The last section describes how the visualization can be navigated and how the navigation is visualized.

In order to analyze the source code it is possible to query the code using a query system. The query system is described in chapter 5. It describes the two query models and gives a description of how to use these query models.

Chapter 6 describes some typical user scenarios for which we describe how they are approached by our visualization tool.

The final chapter, chapter 7, is a refelection on the previous chapters and describes to what end the objectives set out in the introduction are realized. It also gives some points of future work.

Chapter 2

Fact extraction

2.1 Introduction

Fact extraction is the process of processing source code and extracting information (facts) about the source code. These facts can then be used in programs as code analysis tools, source browsers, intelligent editors, automatic document generators or visualization tools.

We are interested in a fact extractor specifically for the purpose of a visualization tool. This visualization tool needs information about the structure of the source code and the semantics of the source code. This information is used in navigating, querying and visualizing the structure. These requirements impose other requirements on the fact extractor. A description of the requirements for the fact extractor is given in section 2.3.

There are a number of existing fact extractors. A few of these fact extractors are described in section 2.4. Advantages and disadvantages are given for each fact extractor with respect to our specific requirements.

The last section, section 2.5, gives some concluding remarks.

2.2 Common terms

Following is a list of common terms that are used throughout the thesis. Most terms come from the field of compiler design. A more detailed description of these terms is provided in [1].

- Abstract Syntax Tree (AST)

  This is a data structure describing source code that has been parsed. A compiler uses this tree as an internal representation of the structure of source code. Abstract syntax trees contain syntactic constructs.

- Abstract Semantic Graph (ASG)

  This is a data structure used in representing or deriving the semantics of source code. The graph is typically constructed from an abstract syntax tree by a process of enrichment (of semantic information) and abstraction. The enrichment may add additional edges to the abstract syntax tree. An example is an edge between an identifier node and a corresponding declaration node.

- Syntactic fact

This is information describing a syntactic language construct. This is usually an instantiation of a production rule from the language grammar [1]. Examples of syntactic constructs are identifiers, iteration statements, selection statements, functions, methods etc.

- Semantic fact

  This is information describing the "meaning" of a particular construct. The information describes type and context related information. Compilers usually contain a phase called the semantic analysis phase which is part of the language parser. In this phase type checking and object binding is done. This phase usually follows the syntactic analysis phase. The semantic information is usually added to the abstract syntax tree. Examples of semantic facts are typing information (i.e. built-in types such as integers and floats, and user defined types) and information that describes the context of a construct such as the linking of identifiers to their declarations and linking declarations to their context (namespace).

- Fact base

  The fact base for a source file is a collection of facts describing the contents of the source file. This can be any kind of information related to the source file but the most common information is information on the syntactic constructs and semantics of the source code.

2.3 Requirements

We are interested in a fact extractor with specific requirements. These requirements are derived from our visualization objectives. Following is a list of requirements for the fact extractor.

1. fact extraction of C/C++ source code

   The language C++ has been chosen because it is a complex language and is widely used in both medium and large scale software projects.

2. fact extraction of any C++ based project

   The fact extractor needs to be able to extract facts from any C++ based project independent of the target compiler for the project. This is because many different projects are built for different target compilers.

3. complete syntactic and semantic fact extraction

   In order to be able to visualize the structure of source code we need complete syntactic extraction as the syntax describes the structure. Semantic facts of the code are needed in order to be able to query the facts. Queries like *"What is the position of the declaration of this identifier"* need semantic information on the type of the identifier in question.

4. mapping of syntactic constructs to source code

   Detailed information on the exact physical location of each syntactic construct is needed. That is, the exact start and end position of the first character and last character of each syntactic construct in (line, column) pairs. This information is required by the structure visualization.

5. platform independent

   The fact extractor needs to be platform independent. This is because clients (visualization tools) that need this fact extractor may run on different platforms.

6. well structured

   The facts need to be structured in such a way that querying and visualizing the facts is easy as this is the primary purpose of this fact extractor.

7. efficient

   To get an insight in the structure of a software project large parts of the project need to be visualized. To this end the fact extractor must be relatively efficient.

We are interested in facts describing the structure and semantics of C++ source code. These facts can be extracted from source code by using a parser for extracting the syntactic structure followed by a semantic analysis process for extracting the semantics of the syntactic structure.

Compilers already syntactically and semantically analyze source code. Writing a compiler for the C++ language is not a trivial task as this language is rather complex. This is why most existing fact extractors are based on an existing C++ compiler. This is described in section 2.4.

2.4 Existing fact extractors

There are a number of existing C++ based fact extractors. The fact extractors differ on the method used in extracting information from source code, the completeness of the extraction and the format used to represent the facts. A brief description on some of the existing fact extractors, namely, GCC-XML [9], CPPX [3] and src2srcml [2] which differ on the afore mentioned points is given.

- GCC-XML is based on the open source GNU Compiler Collection (GCC) C++ compiler front-end. It extracts syntactic and semantic facts (the Abstract Semantic Graph or ASG) from the compiler front-end and stores this ASG in XML format. The XML format used is a direct mapping from the AST (subset of ASG) nested structure to the XML nested structure. The contents of function bodies is not extracted by this fact extractor and therefore the fact extractor is not complete.
- CPPX is, as GCC-XML, based on the GCC C++ compiler front-end. It extracts syntactic and semantic facts (the Abstract Semantic Graph or ASG) from the compiler front-end and stores this ASG as a graph according to the Datrix [6] model in either GXL [7], TA or VCG format.
- Src2srcml extracts syntactic facts and stores these facts in the srcML [2] format. This format preserves the original code formatting and preprocessor specific constructs such as comment, include directives and macro invocations. This fact extractor only partially parses source code. The extractor can be run on incomplete and erroneous source code. No semantic information is extracted.

Both GCC-XML and CPPX are based on the GCC C++ compiler front-end. The advantage of using this compiler front-end is because it provides a complete parser and semantic analyzation stage. Creating a parser and semantic analyzer specifically for the purpose of a fact extractor is a non-trivial task as the C++ language is large and complex. The GCC compiler is open source, in active development and has a large user base. The compiler generates an Abstract Syntax Tree (AST) which contains the syntactic constructs and an enriched version of this AST with semantic information called the Abstract Semantic Graph (ASG). The ASG serves as the input for the GCC-XML and CPPX fact extractors. CPPX extracts the entire ASG. GCC-XML does not extract constructs located in function bodies.

GCC-XML and CPPX do not provide a mapping from the extracted facts to the original source code. Src2srcml preserves the formatting of the original source code and the mapping from the facts to the source code is directly stored in the srcML format.

Because the GCC-XML and CPPX fact extractors are based on the GCC C++ compiler front-end these fact extractors can only extract facts from complete source code. The source code may not

contain any errors and all project related include directories and macro definitions must be properly set in order to be able to extract facts from the source code. This process may be tedious but it results in complete fact bases. The src2srcml fact extractor can extract facts from incomplete source code and/or source code that contains errors. This is an advantage in case that the code contains errors or not all source code is available. However, this results in a fact base that is incomplete.

2.5 Conclusions

None of the fact extractors discussed in section 2.4 completely covers our requirements for a fact extractor.

CPPX completely extracts syntactic and semantic facts but there is no mapping from the facts to the source code. The format used in storing the facts is not suitable for our purposes as we want the facts stored in a format that reflect the structure of the source code.

GCC-XML not completely extracts all facts (no function bodies are extracted) and there is no mapping from the facts to the source code. However, the format used in storing the facts is suitable for our purposes as this is a direct mapping from the source code structure to the XML structure.

Src2srcml not completely extracts all syntactic and semantic facts. However, this fact extractor provides a mapping from the facts to the source code and can run on incomplete and erroneous code. It also extracts preprocessor specific constructs such as include-directives, comment and macro invocations.

We have chosen to design a new fact extractor which completely covers our requirements. The fact extractor is based on the GCC C++ compiler as this compiler offers a complete ASG which may be used to extract syntactic and semantic facts from (as used in GCC-XML and CPPX). The format used to store the facts is similar to the format used in GCC-XML. This format is suitable for representing the structure of the source code. The design of the fact extractor is covered in chapter 3.

Chapter 3

Design of a C++ fact extractor

3.1 Introduction

This chapter describes the design of a C++ based fact extractor. The fact extractor is based on the GCC C++ compiler. It extends this compiler in order to save facts about the syntax and semantics of source code as well as precise information on how these facts are related to the source text itself. These facts are the output of the fact extractor and are stored in an XML formatted file. This chapter is meant as a reference for future work on the fact extractor.

3.1.1 Overview

The general method used for extracting the facts is explained in section 3.2. Following is a section describing the data structure used in the fact extractor. The next three sections, section 3.4, 3.5 and 3.6, give a detailed description of the fact extraction process for the three major processes of the fact extractor. The next section, section 3.7, gives a description of how the fact extractor simulates other target compilers which is needed to be able to run the fact extractor on software projects that are written for different target compilers. Following is a description on how to run the fact extractor and some known problems of the fact extractor in section 3.8. The last section, section 3.9, gives some conclusions about the fact extractor design presented in this chapter.

3.2 Method

This section gives a global description of the method used in extracting facts from source code. We are interested in two types of facts:

- Syntactic facts
- Semantic facts

The *syntactic facts* describe the structure of the source code. These facts are an instantiation of the grammar of the language. The *semantic facts* are important for navigation and analysis purposes as these facts describe type and context related information. The semantic facts are all higher-level information which is, strictly speaking, not directly captured by the grammar of the language itself.

For each of the two data types that need to be extracted we describe four important aspects. The first aspect is determining exactly *what* we want to extract. Next we need to identify *where* we can

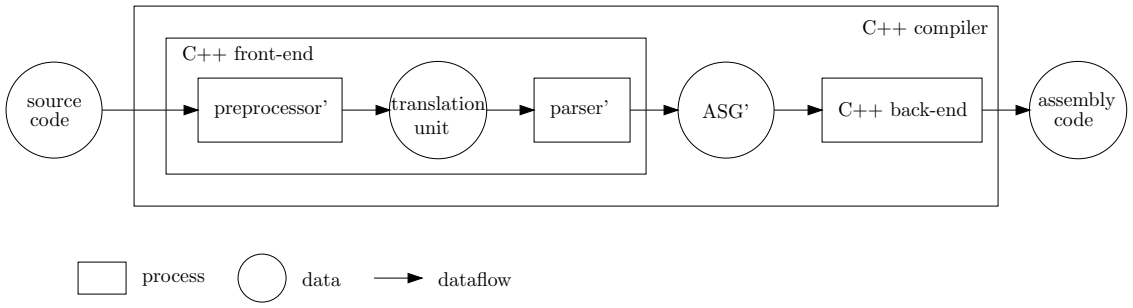


Figure 3.1: Architecture of C++ compiler. The compiler takes source code as input and returns assembly code as output. The compiler can be divided in two sub processes called the front-end and the back-end. The front-end produces an Abstract Semantic Graph (ASG) which serves as the input to the back-end.

extract the facts in the current architecture and *how* we extract the facts. Finally a description is given of *why* a certain method has been chosen.

To be able to describe the aspects we need an understanding of the architecture of the GCC C++ compiler. The architecture of the GCC C++ compiler and the architecture of the fact extractor is described in section 3.2.1. The following two sections, sections 3.2.2 and 3.2.3, give a description of the four aspects related to the syntactic and semantic facts respectively.

3.2.1 Architecture

The architecture of the fact extractor is based on the architecture of the GCC C++ compiler. To understand the architecture of our fact extractor it is important to understand the architecture of the GCC C++ compiler. This section starts with a description of the GCC C++ compiler architecture followed by the architecture of the fact extractor.

GCC C++ compiler architecture

The GCC C++ compiler generates assembly code from C++ source code. The compiler process can be split into two parts. The first part is called the *front-end* which is responsible for preprocessing the source code (handled in the preprocessor process), parsing the source code and semantically analyzing the code (handled in the parser process). The result of this process is the Abstract Semantic Graph (ASG). The second part is called the *back-end* and is responsible for translating the ASG to assembly code. This process is depicted in figure 3.1.

The C++ front-end can be further subdivided in the sub-processes *preprocessor* and *parser*.

The preprocessor translates source code into a data structure, called the *translation unit*. More detailed information about this translation can be found in section 3.4.

The parser generates a parse tree during the parsing process from the translation unit. This parse tree is simplified and is called the Abstract Syntax Tree (AST). The AST will be enriched during the semantic analysis stage with semantic information such as types and declaration locations. This enriched AST is called the Abstract Semantic Graph (ASG).

Fact extractor architecture

The C++ fact extractor builds on top of the GCC C++ compiler architecture. The fact extractor extends the two processes *preprocessor* and *parser* from the GCC C++ compiler architecture. It

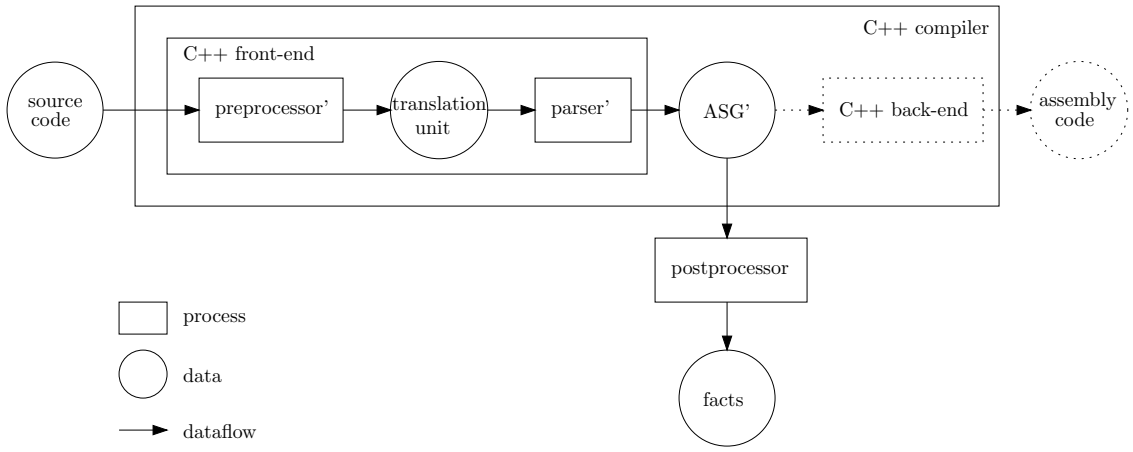


Figure 3.2: Architecture of fact extractor. The architecture is based on the architecture of the GCC C++ compiler front-end. The processes preprocessor and parser are extended and a new process called postprocessor is added which replaces the original compiler back-end.

also replaces the back-end with a new process called the *postprocessor* process. This process has as input a modified ASG called ASG'. This modified ASG is a combination of the original ASG and information gathered from the extended preprocessor and parser processes. This additional information is information on syntactic construct locations, comment and preprocessor directives. This is explained in sections 3.2.2 and 3.2.3. The architecture of the fact extractor is depicted in figure 3.2.

3.2.2 Syntactic facts

The syntactic facts that need to be extracted are closely related to the grammar of the programming language. The grammar is described by a set of production rules. Each production rule's non-literal describes a syntactic element. However, these syntactic elements describe the syntax of the *translation unit*. The translation unit is a transformed version of the original source files. This transformation is performed by the preprocessor. We need to relate the syntactic facts with the original source file's text and not with the translation unit as our visualization builds on top of the source file's textual layout. Most of the syntactic facts can be easily mapped to the original source files as most constructs are not affected by the transformation, however, some constructs are lost due to the transformation process of the preprocessor. The constructs we are interested in are: preprocessor specific constructs such as comment, include-directives and macro invocations together with the constructs derived from the grammar (i.e. the syntactic constructs). The preprocessor specific constructs are important as these constructs appear in the source text and we want to visualize these constructs.

The constructs that are lost due to the transformation of the preprocessor are saved before the transformation is applied. This has been accomplished by extending the preprocessor. The extended preprocessor is denoted in figure 3.2 as *preprocessor'*. The constructs are saved in a temporary list of constructs to be later merged with the final fact base. A detailed description of the method used in extracting preprocessor related facts is given in section 3.4.

The parser process is responsible for identifying syntactic constructs. In this process we extract syntactic constructs we are interested in. We also store detailed location information for each syntactic construct. A detailed description of this process is given in section 3.5.

The constructs extracted from the preprocessor stage can not be extracted at an earlier or later stage in the compiler architecture as at an earlier stage the information is not present and at a

later stage the information is no longer present due to the transformation process.

There are several reasons for extracting syntactic facts during the parsing process. It is at the parsing process that the syntactic facts are identified which means we are not able to extract the facts at an earlier stage of the compiler as there is no information regarding the constructs present yet. However, the parser saves the syntactic constructs in an Abstract Semantic Tree (AST). This tree may be used at an arbitrary point in the compiler's architecture as long as this point is during or after the parsing process. We have chosen to extract facts during the parsing process. There are some reasons why this has been done:

1. There is no one to one mapping between the internal GCC AST construct nodes and the constructs in the source files

   The GCC AST that is built during the parsing process is optimized for the compiler process. A side effect of this optimization is that there is not a direct mapping with the constructs in the tree and the constructs in the source files. To illustrate this case we give an example. The identifier nodes (i.e. nodes describing, for example, variable names) in the GCC AST can not be directly mapped to the constructs in the source files. This is because identifier nodes that represent the same identifier name are represented by a single node in the GCC AST. An illustration is given in figure 3.3. Here a function declaration and a variable declaration have the same name which is represented by the same identifier node in the tree. This is done because the compiler has no need for unique identifier nodes for each construct in the source files which would only require more memory to store. In general syntactic constructs are represented by a group of nodes in the AST. Some of these nodes may overlap with a group of nodes representing a different syntactic construct. However, since we want to couple the syntactic constructs to their physical text representations (for visualization purposes) we do need multiple nodes. A simple solution to this problem is by creating a unique node for each syntactic construct during the parsing process (as an extension to the recursive descent functions [1]). Since the nodes are created during the parsing process we can also directly couple the detailed location information to these constructs. This is explained in more detail in section 3.5.1.

2. Location information needs to be coupled with syntactic constructs during the parsing process

   Since the constructs do not contain detailed enough location information (i.e. line and column positions in the source file) this information must be coupled with the constructs at some stage. It is during the syntactic construct identification in the parser process where information on the location and the constructs is both present. The detailed location information is lost in later stages. This forces us to store detailed location information at the parser process. The recursive descent procedure for every syntactic construct that we want to extract must be extended to couple the detailed location information with the construct. This is explained in more detail in section 3.4.1. Since the recursive descent functions must already be extended to store the detailed location information it requires little changes to further extend these functions to store the syntactic constructs themselves.

3. Recursive descent functions follow construct nesting order

   The recursive descent functions follow the nesting order of the constructs in the source files. This allows us to store the syntactic facts in a tree that is ordered to mimic the nesting order of the syntactic constructs by building the tree during the parsing process. This process is explained in more detail in section 3.5.1. This allows us to build up a tree that is already correctly nested without additional effort.

3.2.3 Semantic facts

The semantic facts that need to be extracted are generated during the parsing process. It is during the parsing process that the syntactic facts are extracted as is explained in section 3.2.2.

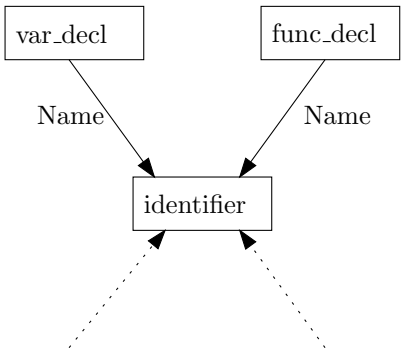


Figure 3.3: Example of identifier node storage in GCC AST

We need to couple the semantic facts to the syntactic facts. We do this by storing a reference to the semantic data during the syntactic fact extraction. This reference is used in a post-processing stage to extract the semantic data and couple this to the syntactic data.

3.3 Data structure

3.3.1 Internal data structure

The general data structure used in the extended C++ compiler front-end to store syntactic and semantic facts is depicted as an UML diagram in figure 3.4.

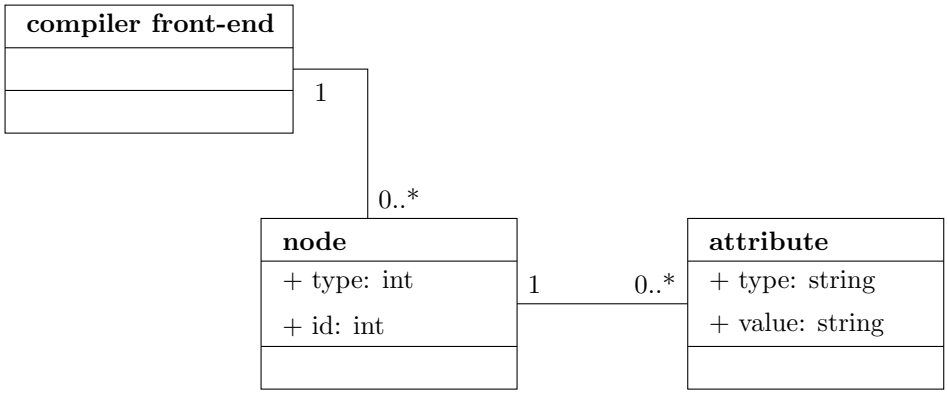


Figure 3.4: UML diagram of datastructure used in extended C++ compiler front-end

The data structure consists of nodes. These nodes represent language constructs such as identifiers, for-loops and declarations but also preprocessor specific constructs such as comment and include-directives. The type of construct is stored in the node as an integer *type*. A unique identifier, *id*, is also stored for node referencing purposes. Each node contains a list of attributes which is used as a general container to store additional information on the construct.

Common additional information stored in the attributes is information which describes the start position and end position of a construct and information on the file which contains this construct. All semantic information such as the type of a construct is also stored as attributes.

Figure 3.5 displays the dataflow of the extracted information between the various processes of the fact extractor. The preprocessor specific constructs (i.e. facts A) are extracted by the *preprocessor'* and the syntactic constructs (i.e. facts B) are extracted by the *parser'*. Facts A and facts B

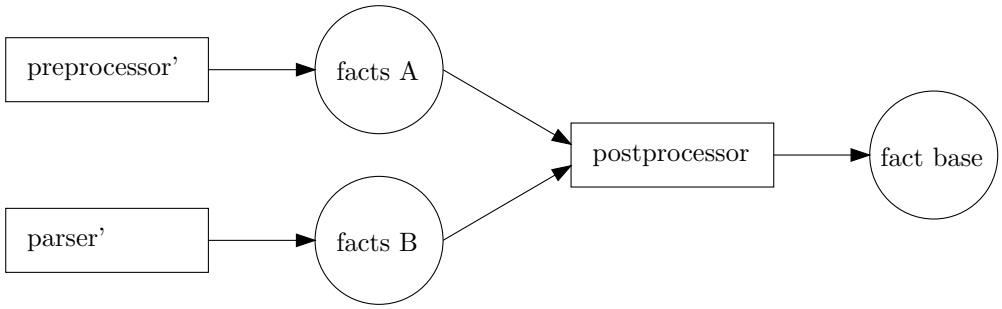


Figure 3.5: Information flow of extracted facts. The preprocessor' generates preprocessor specific facts, facts A. The parser' generates syntactic constructs, facts B. The postprocessor merges facts A and B, enriches these facts with semantic information and produces the final fact base.

are merged together in a single data structure and enriched with semantic information by the *postprocessor*. The result of the postprocessor is the final fact base.

3.3.2 Output format

We have chosen to save the fact extractor's fact base as an XML formatted file. The XML format has been chosen as this format is suitable for exchanging information. The structure of the facts is directly mapped to the tree structure of the XML format. Nodes are represented as tags and attributes are represented as tag attributes.

Following is a minimal example containing the following code fragment:

```
1 int main(int argc, char **argv) {
2   return 0; /* comment */
3 }
```

The facts for this code fragment are stored in an XML formatted file as:

```
<declaration fid="0" location="1:1:3:1">
  <simple_declaration location="1:1:3:1">
    <init_declarator location="1:5:3:1" id="1" kind="function" name="main" returns="int"
                     context="::">
      <identifier location="1:5:1:8" id="2" name="main">
      <parameter_declaration_clause location="1:10:1:30" id="3">
        <parameter_declaration_list location="1:10:1:30" id="4">
          <parameter_declaration location="1:10:1:17" id="5">
            <identifier location="1:14:1:17" type="int" id="6" name="argc"/>
          </parameter_declaration>
          <parameter_declaration location="1:20:1:30" id="7">
            <identifier location="1:27:1:30" type="ptr(ptr(char))" id="8" name="argv"/>
          </parameter_declaration>
        </parameter_declaration_list>
      </parameter_declaration_clause>
      <function_defintion location="1:33:3:1">
        <function_body location="1:33:3:1">
          <compound_statement location="1:33:3:1" id="9">
            <statement location="2:3:2:11">
              <jump_statement location="2:3:2:11" id="10"/>
            </statement>
            <block_comment location="2:13:2:25"/>
```

```
          </compound_statement>
        </function_body>
      </function_definition>
    </init_declarator>
  </simple_declaration>
</declaration>
```

The construct type names in the XML format are listed with their full length name for readability purposes. The actual names used in the XML file are shorter in order to save memory space. The three line code fragment results in a relatively long list of facts. A disadvantage of storing the facts in an XML formatted file is that it requires a significant amount of memory. Parsing the XML format also requires more time as opposed to alternative binary file formats. However, the XML format is suitable for storing the structure of the syntactic facts as this structure can be directly mapped to the structure of the XML format. The XML format is also suitable for data exchange because the XML format is a widely used standard.

A detailed description of the file format is given in appendix C.

3.4 Preprocessor

The C preprocessor transforms source file input to a *translation unit*. This process corresponds to the first four *phases of translation* as described in [8], section 5.1.1.2. These phases are listed in table 3.1 along with a short description of each phase. For a more detailed and complete description of the translation phases, see [8].

Phase	Description
1. Trigraph replacement	Trigraph sequences are replaced by corresponding single-character internal representations.
2. Line splicing	Each instance of a backslash character immediately followed by an end-of-line character is deleted, splicing physical source lines to form logical source lines.
3. Tokenization	The source file is decomposed in preprocessing tokens and sequences of whitespace characters. Each comment is replaced by a one space character.
4. Macro expansion and directive handling	Preprocessing directives are executed and macro invocations are expanded. A `#include` preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively.

Table 3.1: Preprocessor translation phases

Comments, include directives and macro calls located in source files are translated in the translation phases from table 3.1. This translation discards or distorts information on the location of these constructs. No information on the constructs before they are translated is stored in the original preprocessor stage. However, this information is needed by the fact extractor.

This is why phases 3 and 4 need to be extended in order to save information present in the source files before the translation is applied. These extensions are described in sections 3.4.2, 3.4.3 and 3.4.4.

The constructs that need to be saved are stored in a temporary list. This list is later merged with the final data structure. This data structure is described in section 3.3.

3.4.1 Detailed location

The GCC C++ compiler only supports limited information on physical locations of syntactic constructs. This information is present only for diagnostics feedback. However, for our fact extractor we need more detailed information on physical construct locations, and consequently, the preprocessor needs to be extended. The location information is added to the syntactic constructs that are extracted from the parser' process.

$$\ldots, t_n, \underbrace{t_{n+1}, t_{n+2}, \cdots, t_{n+m}}_{s}, \ldots$$

Figure 3.6: Example token stream with syntactic construct s represented by m tokens

The C preprocessor generates a token stream. This token stream is called the translation unit. This translation unit is the input for the parser. The parser analysis the token stream and converts the token stream into syntactic constructs. Each syntactic construct is formed by a number of consecutive tokens from the translation unit. This is depicted in figure 3.6. The start and end position of the syntactic construct is now formed by the start location of the first token and the end position of the last token. In the example the start position of construct s is the first character position of token t_{n+1} and the end position of construct s is the last character position of token t_{n+m}.

The preprocessor already stores information on the start (line, column) position of the first character of each preprocessing token. However, in order to determine the correct end-position of a syntactic construct the lexical scanner and preprocessing token data structure (`cpp_token`) have been extended to store the position (line, column) of the last character of each preprocessing token.

The interface between the parser and preprocessor is formed by a call-back function to the preprocessor. The parser starts its process and requests translation unit tokens from the preprocessor when needed. This means that the parser and preprocessor run in an interleaved fashion. This is different than what is expected from figure 3.1 as it may be expected that these processes are executed one after the other. However the abstraction of the architecture presented in this picture is still valid and gives a more clear representation of the architecture. The interface has been extended to not only pass the token but also the detailed location information. This information is used in the parser process to store start and end location information for each syntactic construct.

The preprocessor also stores the (physical) filename of each preprocessing token. However, this file location information is not detailed enough. The file location does not contain any information of the inclusion order of the respective file. This is important as files with the same name can be included multiple times. Because of conditional preprocessing directives (e.g. `#ifdef, #ifndef, #else`) the contents (preprocessing token stream) of the included files may differ between includes of the same file.

The preprocessor also keeps track of the include stack of a file. However, this information is volatile and is only valid for the current preprocessing token and changes when the file scope changes during the preprocessing stage. This is because this information is only used for diagnostics information and there is no need to store it for later usage. In order to save the include stack of each construct the preprocessor and parser have been extended to save the volatile include-stack information for each syntactic construct in an include-tree. This tree is constructed by merging the various include stacks. A path from a node in the tree to the root of the tree represents an include-stack. Each syntactic construct contains a reference to the correct node in the include-tree which can be used to reconstruct the include-stack of the syntactic construct's file.

3.4.2 Comment

As stated in table 3.1 the comment in source files is translated into a single space character by phase 3 of the preprocessor translation phases. This means that information on comment is not available in subsequent processes. In order to make the information on comment available to later processes the comment specific information is saved in a temporary list.

The comment information contains detailed information on the location of the comment (file location and physical character precise start and end location). Also the type of the comment is saved. We distinguish two kinds of comment, line comment and block comment. Block comment is used for multiline comment sections and line comment is used for single line comment sections. However because the line splicing phase from table 3.1 precedes the tokenization phase line comment may actually span multiple physical lines.

3.4.3 Directives

Line marker directives

The preprocessor concatenates all source code into one stream of code. Internally the source code is numbered with logical line numbers. These line numbers do not correspond with the physical line numbers. To be able to access the physical line number (for diagnostics information - and in our case location information extraction) a mapping m is maintained which allows for requesting a physical line number given a logical line number.

Interpretation of the line command and line marker (`#line num` and `# num "file" [flags]`) preprocessor directives changes the mapping m from logical line numbers to physical line numbers to the line numbers and filenames stated in the line command and/or line marker. This is good for diagnostic feedback but it removes the possibility of requesting the actual physical source location of subsequent syntactic constructs.

To prevent the mapping m to change when a line command and/or line marker is processed a call to the actual function that changes the line mapping has been removed from the preprocessor. This means that the preprocessor still processes the directives but it does not change m anymore. Diagnostic messages now display the actual physical location of the feedback rather then the location set by the directives.

The location of all subsequent syntactic constructs can now be requested from mapping m.

Include directives

Include directives (`#include`) causes the named header or source file to be processed from phase 1 through phase 4, recursively as stated in table 3.1. However, information on the actual location of the include directive is not saved.

As an include directive is also a syntactic construct the preprocessor has been extended to save information on the include directives.

3.4.4 Macros

Macro invocations are expanded by step 4 of the preprocessor translation phases as listed in table 3.1. GCC does not store any information regarding this invocation in the data structure. Only the expanded tokens are saved for later usage in the parsing process. However, since we are interested in visualizing the source code as it is before the translation phases we need information on the location of the macro invocations. This information can be extracted from the preprocessor by extending the function that handles the macro expansion. The preprocessor provides a stream

of preprocessing tokens. Other processes request tokens from the preprocessor. The preprocessor silently expands macro invocations when encountered. The macro expansion is implemented by a non recursive function. This means the entire macro invocation will not be expanded in one pass. Instead the macro is expanded into a new set of tokens. These tokens may again contain other macro invocations. The new set of tokens is stored in a temporary token buffer. When another process requests a token from the preprocessor it first handles tokens from the temporary buffer if this buffer is not empty. When a token from this buffer is a macro invocation it again expands this macro and adds the expanded tokens to the temporary buffer.

The expansion tokens that result from the macro invocation are of no interest for our visualization needs. However, we need these tokens for our fact base to be complete. This is important as our query system might contain queries that analyze macro specific code. An example of such a query would be "what is the largest defined macro?". Furthermore, these tokens can not simply be removed from the process as they are needed by the compiler in order to correctly handle the parsing process.

The locations of the constructs that result from the macro expanded tokens have no meaning as the constructs are generated by the preprocessor and are not physically present in the source code. However, we still want to provide these construct in our fact base to make the fact base complete. We set the start and end locations of the macro expanded constructs to the start and end locations of the macro invocation that generated these constructs.

We implemented the assignment of locations to the macro expanded constructs by assigning the character start position of the macro invocation to every preprocessor token that was generated by the macro invocation. We do the same for the end position. This is valid as the final start and end positions of the syntactic constructs are formed by taking the start position of the first token that forms this construct and by taking the end position of the last token that forms this construct. This guarantees that the macro expanded constructs will have the same location as the macro invocations.

We store in a separate list the location of every macro invocation encountered. This list is later merged with the reduced abstract syntax tree that is constructed during the parsing process. When a sub tree is encountered during the merging process that has the same physical location as the macro invocation we know that this sub tree is the result of that specific macro invocation. We store the macro invocation node as the parent of this sub tree. This allows the visualization process to easily identify macro invocation generated sub trees and can skip this sub tree when desired.

3.5 Parser

3.5.1 Constructing reduced abstract syntax tree

The GCC C++ parser builds a compiler-optimized Abstract Syntax Tree during the parsing process. We build a new reduced Abstract Syntax Tree (AST) that meets our specific requirements. The tree is reduced in the sense that it does not contain all syntactic constructs present in the compiler generated AST. An example of constructs that are not present in the new AST are compiler generated (artificial) constructs such as implicitly generated class constructors and compiler specific internal defines.

The parser is of the recursive descent type [1]. We build the new AST by extending the recursive descent functions. The recursive descent functions are called by the parser to parse each syntactic construct. We extend the function by storing a syntactic construct node corresponding to the type of construct that is parsed. We also extend the recursive descent functions to store the detailed location of each construct. This location information is stored with the node in the new AST.

The parser calls the recursive descent functions according to the grammar description. This means that the functions are called in such a way the structural nesting is followed. However as listed

in the comment of the parser code we can read that the parser needs a look-ahead to allow for disambiguating certain C++ constructs. Therefore the parser is capable of parsing "tentatively". When the parser is not sure of what construct comes next it enters this mode. Then, while attempting to parse the construct, the parser queues up error messages, rather than issuing them immediately, and saves the tokens it consumes. If the construct is parsed successfully, the parser "commits", i.e., it issues any queued error messages and the tokens that were being preserved are permanently discarded. If, however, the construct is not parsed successfully, the parser rolls back its state completely so that it can resume parsing using a different alternative.

We construct the new AST by keeping track of the current context in the AST. This context is the current position represented by a pointer to a node in the new AST. At the start of the recursive descent function the start location of the construct is saved and a node, n, is added to the new AST in the current context. The new context will be set to point to node n. If the parsing of the construct fails due to an error or due to tentative parsing the parser rolls back. If this happens we remove node n and set the current context to point to the parent of node n. If no parsing error occurs the end location of the construct is saved to node n and the node is permanently stored in the AST. The context is now set to the parent node of node n.

Following is a piece of pseudo-code which lists the general method used in extending the recursive descent procedures.

```
void recursive_descent_procedure() {
  save_start_location(start_loc);
  push_node_context(NODE_TYPE, start_loc);

  ...

  if (rollback) {
    shred_node_context();
  } else {
    save_end_location(end_loc);
    pop_node_context(NODE_TYPE, end_loc);
  }
}
```

In theory this process will create an AST which is correctly nested according to the construct nesting order. However, in practice this is not the case. This is explained in section 3.6.3.

We also add a reference to the corresponding node in the GCC generated Abstract Semantic Graph (ASG). This reference is used in the postprocessing stage to extract additional syntactic and semantic data. This is explained in section 3.6.

3.5.2 Type extraction

We build a new AST during the parsing process as explained in section 3.5.1. References to nodes in the GCC generated ASG are stored in the nodes of this tree. These references are used to extract additional syntactic and semantic data from the data structure generated by GCC. However, in some cases this reference is not enough to allow for extracting type information in the postprocessing stage. An example is the extraction of identifier nodes. We need to couple the type of the identifier with the identifier node in the postprocessing stage. However, this node only contains a reference to a node in the GCC generated ASG. The identifier node in the GCC ASG is an end node. This means that there are no directed edges pointing from this node to other nodes. This is because identifier nodes are not unique. An identifier node in the GCC ASG may represent multiple identifiers with the same name in the source code. This is depicted in figure 3.3. In order

to be able to link the type information with these nodes some recursive descent functions have been further modified to explicitly request the type from the parser given the current context and are added to the nodes as special type-pointers.

Another complication is that during the parsing process GCC builds the AST but also constructs the ASG by extending the AST with type information. In some situations it is impossible to request the type of a construct during the parsing process because the type information is not yet added to the AST. This is, for example, the case for identifiers located in the argument list of a member declaration. The request for the type of the identifier located in the recursive descent procedure for identifiers fails. This is because scanning the complete argument list of the member declaration is not finished at the time the recursive descent procedure for the identifier is called and as such the ASG does not contain type information for the argument identifiers. The solution that has been chosen is to request the type of the identifiers after the scanning of the argument list has been finished.

3.6 Postprocessor

3.6.1 Abstract syntax data extraction

During the parsing phase of the fact extractor nodes corresponding to syntactic constructs are added to a new reduced abstract syntax tree. These nodes contain a pointer to the corresponding node stored in the GCC generated ASG. The referenced nodes contain additional information on the syntactic construct. This information is extracted in the postprocessing stage as to minimize the code change in the parser code itself.

As an example we have nodes for syntactic constructs of the type *iteration statement*. However, the grammar also describes production rules for the type of iteration statement (`for`, `while` or `do`) and as such this information is part of the syntax. The additional syntax information is stored in the original GCC node. We extract this information from the GCC nodes and add this information as attributes to the node as stored in the reduced AST.

3.6.2 Abstract semantic data extraction

The nodes in the newly created AST, i.e. AST', contain pointers to corresponding nodes in the GCC generated ASG. Type information on these nodes is extracted from the ASG by using the pointer to the node in the ASG. The referenced node contains edges to corresponding type information. This type information is extracted and added to the newly generated AST as attributes.

In some situations the reference to the node in the ASG can not be used to extract type information. This has been explained in section 3.5.2. For these situations an additional type-specific reference is available to the corresponding type node in the GCC generated ASG. This reference is used to extract the required type information and this information is added to the newly generated AST as attributes.

3.6.3 Containment relation

Every node in AST' may have zero or more child nodes. Each child node is a sub-construct of it's parent node. This means that the string corresponding to the child node is a sub string of the string corresponding to the node. This is called the containment relation.

The parser sometimes deviates from the expected parsing order. Some constructs are saved in a token buffer to be scanned later in the parsing process. This, for example, is the case for inline class functions. The function bodies are saved and are only parsed after the top class has been

parsed completely. Because of the way we construct the new AST the containment relation is violated.

We handle these special cases in a postprocessing stage in order to restore the proper containment relation of the constructs in the tree.

3.7 Compiler simulation

Source code written for a particular compiler is not always compatible with a different compiler even when the compilers are both based on the same C++ standard. There are slight differences between different target compilers which causes these incompatibilities. Following is a short list of the main points which cause the incompatibilities.

1. Compiler specific extensions to the C++ standard
2. Non-strict conformance to the C++ standard
3. Compiler specific predefined macros
4. Compiler specific include files

All of the above listed points causes source code written for a specific compiler to be incompatible with a different compiler. The goal of the fact extractor is to be able to parse any valid C++ source code. That means that code not specifically written for the GCC C++ compiler must also be parsed without any problems. Making it possible to parse source code initially written for a different target compiler is called *simulating* a compiler. There is not a general way to simulate *any* compiler by the GCC C++ compiler as each compiler has its own specific, usually small set of, differences.

For each different target compiler that must be supported by the fact extractor the above four points must be addressed. This has been done for the Microsoft Visual Studio 6.0, 7.1 and 8.0 compilers. The GCC dialect is, of course, supported by definition. Following are four subsections which describe the process involved in simulating the Microsoft Visual Studio C++ compilers with the GCC C++ compiler. These sections follow the four points listed above. Each section starts with a problem description and ends with a solution. Simulating different compilers is done using a similar set of solutions and these sections can be used as a good starting point.

3.7.1 Compiler specific extensions to the C++ standard

The Microsoft Visual Studio C++ (MSVC) compilers have a number of extensions to the C++ standard which conflict with the GCC C++ compiler.

Additional keywords

The standard has been extended by MSVC with a sized integer fundamental type. The type notation is `__int`n, where n is the size of the integer in bits. The value of n can be 8, 16, 32 or 64.

The "new" fundamental types are simulated using macro definitions that use existing fundamental types. These macro definitions are listed in table 3.2.

Different calling convention syntax

Most compilers provide several different conventions for calling internal or external functions. However, the syntax of these calling conventions is not in any standard and each compiler has its own syntax. This means that source code that uses calling conventions for a particular compiler is not compatible with another compiler. The differences in notation are usually minimal.

Since each compiler has its own syntax it is not possible to provide a general way to support all these different notations. For now, only the Visual Studio C++ compiler versions 6.0, 7.1 and 8.0 are supported. Following is a short description on the differences between the Visual Studio C++ compiler notation and the GCC C++ compiler notation.

When making a declaration for the GCC C++ compiler it is possible to specify special attributes using the `__attribute__` keyword. This keyword is followed by an attribute specification inside double parentheses. The attribute specification name may be specified with __ preceding and following each keyword. This is done to prevent name clashes with macros (with the same name) defined in header files.

The MSVC notation does not use the `__attribute__` keyword nor the double parenthesis following this keyword. The attributes are added to the code by using their respective name.

The fact extractor only supports the calling convention syntax of the GCC C++ compiler as this is the base of the fact extractor. To simulate the calling convention syntax of the MSVC compiler some "translation" macros are fed to the fact extractor. These macros add the `__attribute__` keyword followed by the name of the attribute enclosed in double parenthesis for each attribute name. The calling convention specific macros are listed in table 3.2.

Calling conventions
__thiscall=__attribute__((__thiscall__))
__stdcall=__attribute__((__stdcall__))
__cdecl=__attribute__((__cdecl__))
__fastcall=__attribute__((__fastcall__))
_stdcall=__attribute__((__stdcall__))
_cdecl=__attribute__((__cdecl__))
_fastcall=__attribute__((__fastcall__))
__declspec(x)=__attribute__((x))
Fundamental types
__w64= __int64="long long"
Predefined macros
__cplusplus _MSC_VER=1310 _MSC_EXTENSIONS _WIN32 _WIN64 _M_IX86 _M_IA64 _WCHAR_T_DEFINED _NATIVE_WCHAR_T_DEFINED
Miscellaneous
_inline=inline __forceinline=__inline PASCAL= RPC_ENTRY= SHSTDAPI=HRESULT _INTEGRAL_MAX_BITS=64 __uuidof(x)=IID() SHSTDAPI_(x)=x

Table 3.2: Visual Studio 6.0, 7.1 and 8.0 simulation macros

3.7.2 Non-strict conformance to the C++ standard

The MSVC compilers do not strictly follow the C++ standard. The compiler has an *implicit typename extension*. The extension allows for missing `typename` keywords in source code where

they can be implicitly derived from the context. This implicit typename extension was also present in the GCC C++ compiler prior to version 3.4.0. However, at the time the C++ parser has been rewritten to a manually written recursive descent parser in version 3.4.0 the decision has been made to drop the implicit typename extension. This decision was made because the implicit typename extension had a number of bugs and ambiguities and did not conform to the C++ standard. Following is example code which illustrates the implicit typename.

```
template<class T>
T A::get(void) {
  vector<int> v;
  v::iterator it; // incorrect in GCC C++ 3.4.0+
  typename v::iterator it; // correct in MSVC and GCC
  ...
}
```

Non strict conformance to the C++ standard introduces compatibility conflicts. In some compiler a certain notation (in this case the lack of the `typename` keyword) is allowed while in another (more strict) compiler this is not allowed.

The compiler used in the fact extractor does not have an implicit typename extension. This means that the source code that is written for a compiler that does have an implicit typename extension can not be parsed by the fact extractor. There is no way to circumvent this problem by means of compiler switches. The solution is to patch the source code by means of writing the `typename` keyword at the appropriate places. The source code that needs to be patched may include user-code and compiler specific include files. Patching the user-code is a problem since this is the responsibility of the end-user of the fact-extractor. Patching compiler specific files can be done beforehand and these patched files can be added to the fact-extractor. Patching compiler specific include files is explained in section 3.7.4.

3.7.3 Compiler specific predefined macros

Each compiler has a set of predefined macros, i.e. compiler built-in macros. Since these macros may differ between different compilers this introduces incompatibility issues.

The intersection of the set of GCC C++ compiler predefined macros and the set of MSVC C++ predefined macros do not cause any problem since these macros are available in both compilers. In this set the ANSI compliant predefined macros (macros such as `__DATE__`, `__FILE__`, `__LINE__` and `__TIME__` - for a complete list see section 16.8 of the C++ standard [8]) are usually present.

However the macros that are available in the MSVC compiler and *not* in the GCC C++ compiler cause a problem since the GCC C++ compiler must simulate the MSVC compiler. These macros can be easily simulated by the GCC C++ compiler by supplying these macros at the command line of the compiler. A number of these predefined MSVC macros are listed in table 3.2.

3.7.4 Compiler specific include files

Compilers usually come with their own set of include files. In case of the MSVC compilers an interface to the standard template library (STL) and platform software development kit (platform SDK) are included with the compiler. These header files are written for the MSVC compiler and have the same problems as described in the previous sections. Some of these problems are solved by supplying the GCC C++ compiler with correct "simulation" macro definitions. However, the implicit typename problem can not be solved by compiler switches.

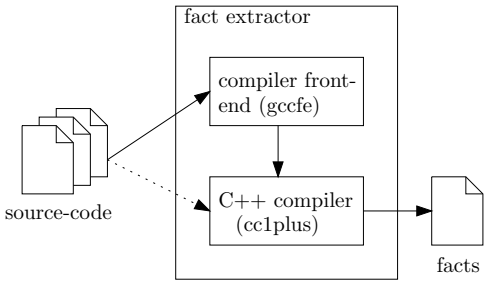


Figure 3.7: fact extractor executable graph

The MSVC specific header files need to be patched such that the GCC C++ compiler can correctly parse them. However, the original header files must be preserved such that the MSVC compiler will not use the patched header files. This has been solved by saving the patched header files in a separate directory. This "patch" directory is then supplied to the GCC C++ fact extractor. This directory is a special include directory since the files in this directory must override the original (unpatched) files (i.e. they have always precedence over other include files).

The GCC C++ fact extractor has been extended with a new switch that allows for supplying the compiler with the special include directory. This switch is called the *include wrapper* switch as the include directory "wrappes" the original (unpatched) directory. This switch is also present in the gccxml [9] fact extractor but the implementation differs because the source code architecture regarding include directories (include chains) management has been significantly changed between the GCC C++ version used by gccxml (v3.3.2) and the GCC C++ version used by this fact extractor (v3.4.2).

3.8 Running

The fact extractor consists of two executables.

gccfe This executable is the main executable for the fact extractor. This executable serves as a front-end for the compiler (`cc1plus`) executable.

cc1plus This is the modified GCC C++ compiler. It contains the processes as depicted in figure 3.2.

The preferred way to execute the fact extractor is by running the compiler front-end `gccfe`. This executable searches the system for an installation of a supported compiler. If such a compiler has been found it sets the correct include-directories, patch-directories and compiler simulation macros. It also sets some common compiler switches as described in table 3.3. Valid options for `gccfe` are described in table 3.4.

Another way of executing the fact extractor is by running the compiler (`cc1plus`) executable directly. However, this means that all compiler simulation switches and general compiler switches must be provided manually.

Both ways of executing the fact extractor are depicted in figure 3.7.

3.8.1 Known problems

The fact extractor is based on the GCC C++ compiler version 3.4.2. This compiler more strictly follows the C++ standard which introduces some problems as described in section 3.7.2. The

switch	description
`-quiet`	do not display functions compiled or elapsed time
`-fsyntax-only`	this forces the compiler to only check for syntax errors and then stops (does not run the original compiler back-end)
`-w`	suppress all warnings
`-o NUL`	no output files other than the structure information file are created
`-nostdinc`	does not search standard system include directories

Table 3.3: Common compiler switches

switch	description
`--gcc-executable <xxx>`	Set the executable to use; default: cc1plus_struct.exe
`--gcc-compiler <xxx>`	Set the compiler version to simulate; valid: MSVC60, MSVC71, MSVC80, AUTO, CUSTOM; default: AUTO
`--debug`	prints verbose output to stdout

Table 3.4: Valid `gccfe` switches

switch	description
`-iwrapper <path>`	adds include file directory *path* to the `iwrapper` include chain
`-fxml=<file>`	sets the name of the structure output file, the default value is `structure.xml`
`-fxmllocalscope`	only generates non-recursive (no includes) structure information

Table 3.5: Additional compiler command-line switches

include files for various target compilers can be patched to conform to this more strict compiler. However, it may be possible that the user code also has to be patched to conform to the standard. This is a problem since it is now up to the end-user of the fact extractor to patch the code. An end-user may not have the knowledge or expertise to patch the code to conform to the standard. This problem is inherent to the used parser and can not be solved.

3.9 Conclusions

Designing and implementing the fact extractor has been a time consuming process. This is because there was no prior knowledge about the GCC C++ compiler and because the compiler is a large project that is poorly documented. The process of gaining an understanding of the code has taken a significant amount of time.

The method of extracting facts presented in this chapter is not an optimal solution. There are disadvantages of extracting the facts during the parsing process. This is because the parser does not follow the exact nesting structure of the constructs in all cases as is described in section 3.6.3. Also, the semantic facts are hard to extract since we extract references to semantic data during the parsing process where not all semantic information is present yet. Solutions to these problems have been presented but it is difficult to ensure that all cases have been covered. Another method would be to completely extract syntactic and semantic information from the GCC ASG as a postprocessing stage. This will make extracting the semantic information much easier and also ensures that all information is extracted. However, this has as disadvantage that it is difficult to map the detailed location information with the syntactic constructs. This mapping of the location information with the syntactic constructs is robust in the presented method and is one of its major advantages.

Choosing to built a fact extractor instead of using a fact extractor such as GCCXML or CPPX is not ideal but preferable to using GCCXML or CPPX. This is because we need location information on the constructs for our visualization purposes and this information is not provided by any of these tools (except for src2srcml). Src2srcml provides location information but does not provide semantic information which is needed for our code analysis (i.e. query-system).

It may be clear that no method is ideal and choosing some method over another method results in advantages in some area but disadvantages in another.

Chapter 4

Visualization

4.1 Introduction

Our goal is to insightfully visualize the structure of C++ source code. To visualize this structure we distinguish three types of data. On the highest level we have a set of *files*. Each file contains source code, the *text* of the file. The text structure is made up of various syntactic *constructs*.

For the three types of data (i.e., files, text and constructs) we distinguish four important aspects with respect to the visualization. These aspects are *how to visualize?*, *how to layout the visualization?*, *what to show in the visualization?* and *how to navigate?*.

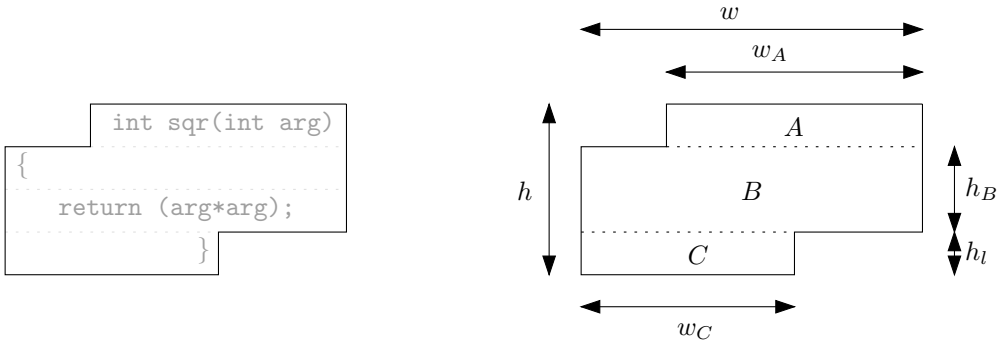
This chapter is subdivided into sections about the afore mentioned four aspects of visualization. Each section explains the aspects in relation to the three types of data. The four aspects are described in the four sections "Visualization method" 4.2, "Layout" 4.3, "Visibility" 4.4 and "Navigation" 4.5.

Most of the sample source code in this chapter is taken from the Visualization ToolKit (VTK) [12]. This toolkit is an open source software system for 3D computer graphics, image processing, and visualization and consists of over 700 C++ classes and over 350,000 lines of code. Because of the large code base this toolkit serves as a good sample software project for the sample images in this chapter.

4.2 Visualization method

4.2.1 Syntactic construct shape

Each syntactic construct is actually a string of text in a source file. We visualize these constructs by drawing a texture in the background of the text shaped according to the contour surrounding the text string. This contour is displayed in figure 4.1 (a). This contour has a general shape which is displayed in figure 4.1 (b). The general shape can be subdivided in three regions. A top rectangle A, a body rectangle B and a bottom rectangle C. The height h_l of the top and bottom rectangles are identical since these rectangles correspond to the leading and trailing lines of the construct. The height h_B of the body rectangle B is a multiple of the height h_l of the leading and trailing lines, i.e. $h_B = i * h_l$, for all $i \in \mathbb{N}$.



(a) (b)

Figure 4.1: (a) Construct contour; (b) General construct shape

(a) (b)

Figure 4.2: (a) Constructs visualized by contours; (b) Constructs visualized by shading

4.2.2 Construct visualization

Visualizing the nesting structure of the constructs by drawing the contours of the constructs as outlines does not always give a clear clue about how the constructs are nested. Usually the constructs are nested in such a way that each sub construct is delimited by a margin. This is a direct consequence of applied coding standards which usually require the programmer to correctly indent code and insert white lines before various constructs. Constructs that have a body height h_B greater than zero may be extended to the left and right which increases the construct's width w and adds a margin to the constructs that are contained in this construct. However, even with these margins the nesting structure is not clearly visible when drawing hundreds of constructs. Changing the colors of the borders helps to discriminate various constructs but again does not give a clear picture of how the constructs are nested.

We use shading to visualize the nesting structure. This method is similar to the method used in [13]. Figure 4.2 (a) shows that the nesting structure of constructs visualized by drawing the construct's contours is less clearly visible than when we use shading techniques to visualize the constructs as depicted in figure 4.2 (b).

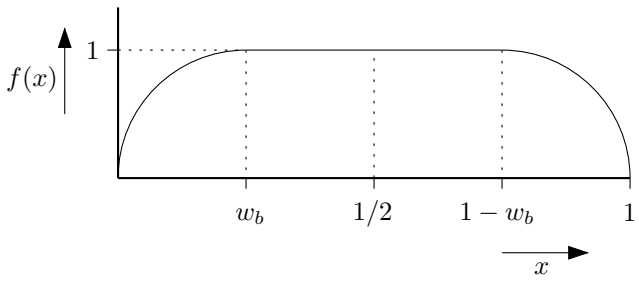


Figure 4.3: Two dimensional cross-section of construct height profile

4.2.3 Method

We use shading to visualize the constructs. The shading is interpreted to be an illuminated surface. Hence, we transform the constructs into a three dimensional profile. The planar dimension displays the construct's shape and the shading displays the construct's height.

The method as used in [13] uses parabolic height functions as a basis for drawing a rectangular shaped cushion like 3D profile. In our case the shape of the construct is determined by the shape of the corresponding text string. Because it is possible to display the text in the cushion like profile there is a close connection between the text of the construct and the actual construct's visualization. The shape of the construct is planar, however when using parabolic functions as the basis for the cushion profile the resulting shading distorts the intuitively planar representation of the construct. To minimize this distortion but still have the advantages of the shading to visualize the nesting we use a height function f that partly has a curvature shape, called the border, followed by a constant height section. The width of the border section, w_b, is a parameter to the height function f. The width w_b is expressed as a fraction of the total width of the cushion. We call these cushions "flattened cushions".

For simplicity we first define the two dimensional height profile f (cross-section) of the flattened cushions on the domain $[0, 1]$. Next we define, given the two dimensional height profile f, the three dimensional height profile of square shaped flattened cushions on the domain $[0, 1] \times [0, 1]$ followed by rectangular shaped cushions on the domain $[0, w] \times [0, h]$. Next we describe the method used in constructing non-rectangular shaped cushions using the rectangular shaped cushions as their basis.

The height function f is given in equation 4.1 and is graphically depicted in figure 4.3.

$$f(x) = \begin{cases} g(\frac{x}{w_b}) & \text{for } x \in [0, w_b\rangle \\ 1 & \text{for } x \in [w_b, (1 - w_b)] \\ g(1 - \frac{1-x}{w_b}) & \text{for } x \in \langle(1 - w_b), 1] \end{cases} \tag{4.1}$$

The equation f is valid for values $w_b \in [0, 1/2]$ and $x \in [0, 1]$.

The function g describes the curvature shape of the construct's border section. The actual function used for g is discussed in section 4.2.5 as there are several advantages and disadvantages for various different functions.

The 3D height profile is derived from the 2D height profile by function $h(x, y)$ given in equation 4.2.

$$h(x, y) = f(x)f(y) \qquad \text{for } x, y \in [0, 1] \tag{4.2}$$

The resulting 3D height profile has a cushion like profile with a flat top, the size of which depends on the value of w_b. The height of the cushion is zero at the contour of the cushion and has a maximum height value of one.

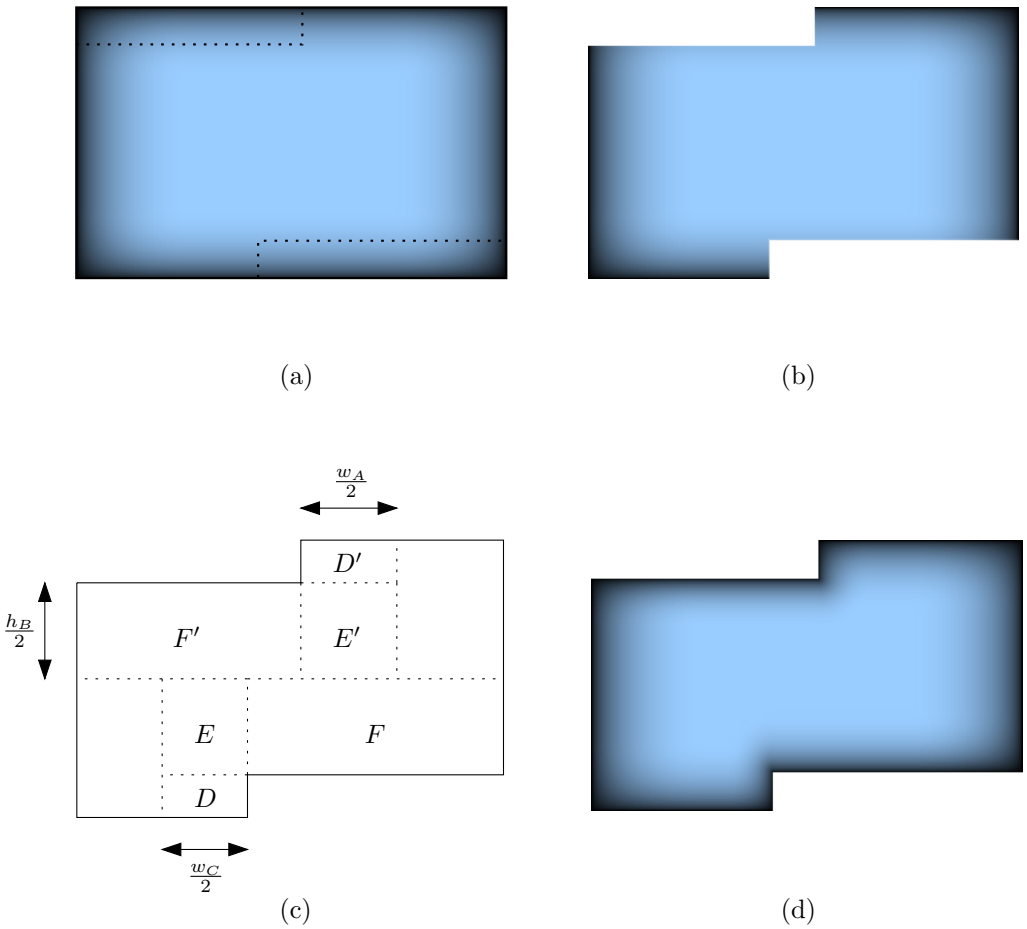


Figure 4.4: Cushion construction process. In step one the cushion bounding box is drawn (a) followed by removing sections not part of the construct shape (b). Next the cushion is subdivided (c). The areas in the subdivision are scaled down to form the final cushion rendering (d).

The rectangular shaped cushions are derived from this equation by stretching the square shape $[0,1] \times [0,1]$ over the rectangular shape $[0,w] \times [0,h]$. A function for this stretched profile is given in equation 4.3. An example of the resulting profile is depicted as a shaded image in figure 4.4 (a).

$$h_s(x,y) = h(x/w, y/h) \qquad \text{for } x,y \in [0,w],\,[0,h] \wedge w \neq 0 \wedge h \neq 0 \tag{4.3}$$

The method of constructing the general non-rectangular shaped (see figure 4.1 (b)) cushions is strongly influenced by the efficiency requirements for the cushion rendering. These requirements are described in section 4.2.4. The process of constructing a general non-rectangular cushion shape can roughly be divided in four steps:

a. Construct a 3D height profile of the rectangular shaped bounding box of the cushion shape. This is done by using equation 4.3. This is depicted in figure 4.4 (a).

b. Remove the portions of the height profile from the bounding box that do not belong to the general cushion shape. This is depicted in figure 4.4 (b). After this step we have a 3D profile that does not have a height of zero at all edges of the shape's contour.

c. To fix the incorrect height at some edges we subdivide the general cushion profile in six

subsections called D, E, F, D', E' and F' as displayed in figure 4.4 (c). These sections all share a border with the edges of the contour that have an incorrect height value.

d. We scale down the height profile in the six sections D, E, F, D', E' and F' in order to restore that all contour edges have height zero. The final geometry is depicted as a shaded image in figure 4.4 (d).

The final height profile function $H(x, y)$ for the general shape cushion is given by equation 4.4.

$$H(x,y) = \begin{cases} h_s(x,y) & \text{for } (x,y) \notin D \cup E \cup F \cup D' \cup E' \cup F' \\ h_s(x,y)s_X(x',y') & \text{for } X \in \{D,E,F,D',E',F'\} \wedge (x,y) \in X \end{cases} \tag{4.4}$$

The X in the scaling function s of equation 4.4 is one of $\{D, E, F, D', E', F'\}$ depending on in which subsection the coordinate (x, y) is located. Each subsection has its own scaling function s_X. The coordinate (x, y) is transformed to a normalized coordinate (x', y') for the subsection X in which the coordinate (x, y) is located. The lower left corner of each subsection has normalized coordinate $(0, 0)$ and the top right corner has normalized coordinate $(1, 1)$. The scaling functions use the same function f to scale down the geometry as described in equation 4.1. However, only the first halve of this profile is used. This is because we want the scaled down geometry to closely resemble the geometry of the non scaled down border sections. The first halve of the function f is represented by function k as given in equation 4.5.

$$k(x) = f(x/2) \qquad \text{for } x \in [0,1] \tag{4.5}$$

The scaling functions s_X for each subsection X using normalized coordinates are described in equations 4.6 through 4.11.

$$s_D(x',y') = k(1-x') \tag{4.6}$$

$$s_E(x',y') = \begin{cases} k(|(x',y')-(1,0)|) & \text{for } |(x',y')-(1,0)| \leq 1 \\ 1 & \text{for } |(x',y')-(1,0)| > 1 \end{cases} \tag{4.7}$$

$$s_F(x',y') = k(y') \tag{4.8}$$

$$s_{D'}(x',y') = k(x') \tag{4.9}$$

$$s_{E'}(x',y') = \begin{cases} k(|(x',y')-(0,1)|) & \text{for } |(x',y')-(0,1)| \leq 1 \\ 1 & \text{for } |(x',y')-(0,1)| > 1 \end{cases} \tag{4.10}$$

$$s_{F'}(x',y') = k(1-y') \tag{4.11}$$

The scaling equations for subsections D', E' and F' are rotated versions of the scaling equations for subsections D, E and F respectively. To make sure the shading is continuous all the edges that are shared among the six sections D, E, F, D', E' and F' must have the same scaling profile. The scaling functions of sections E and E' must have the profile of function k at two of its borders. These borders share the same corner point. The profile is distributed radially over the square base to ensure continuous shading.

The general shape does not necessarily have a connected contour. When the height h_B of the body B is zero and the combined width of the rectangles A and C is less then the total width w, i.e. $w_A + w_C < w$, given that $w_A > 0$ and $w_C > 0$ the shape consists of two rectangles. This case is displayed in figure 4.5. The cushion is constructed as if it were two separate rectangular shaped constructs that we may simply construct using equation 4.3.

The method in [13] sums up parabolic functions for visualizing the nesting structure. We can use this same method in our case, this is visualized in figure 4.6 (a). However, since it is common that there is enough margin surrounding each construct and the border width w_b is relatively small the

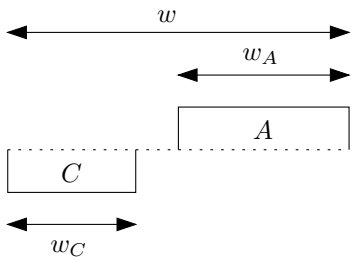


Figure 4.5: Disconnected contour shape

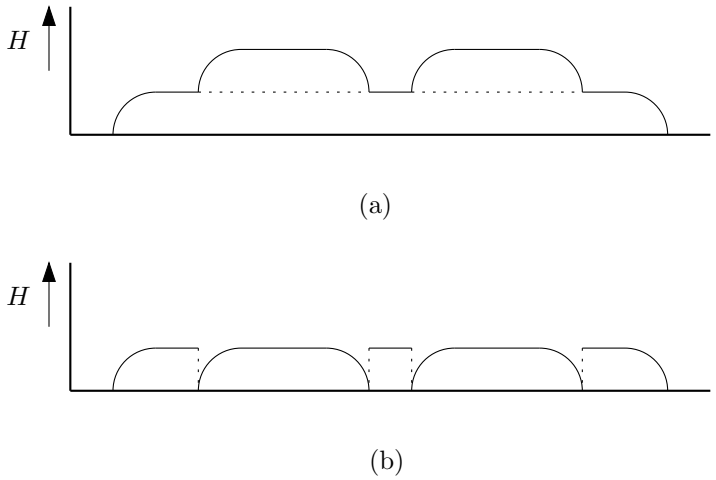


Figure 4.6: Construct nesting by summing (a) and replacing (b)

sub construct cushions commonly lie within the flat area of their parent cushions. This allows us to "override" each construct's geometry with the geometry of the deeper nested constructs as is depicted in figure 4.6 (b) without affecting the final shaded image. This is because the slopes of the latter profile do not change with respect to the slopes of the "summed up" geometry profile.

4.2.4 Implementation

One of the requirements is that the cushion rendering is efficient. Operations such as zooming, scrolling and changing profiles must be performed in real-time using a modern PC and graphics card. We have chosen to render the cushions using OpenGL [11] with common texture operations.

In the cushion treemaps visualization [13] shading the geometry is done using equation 4.12.

$$I = I_a + I_s \max(0, \frac{\mathbf{n} \cdot \mathbf{l}}{|\mathbf{n}||\mathbf{l}|}) \tag{4.12}$$

where I_a is the ambient light intensity, I_s is the intensity of the directional light source, $\mathbf{n}$ is the normal of the surface and $\mathbf{l}$ is a vector that points towards the light source.

We shade the height profile H as given in equation 4.4 by directly mapping the height value of the profile to the intensity value I as given in equation 4.13:

$$I = I_a + H(x, y) \tag{4.13}$$

where I_a is the intensity of the ambient light.

This method has as advantage that when stretching the geometry of the cushion the slope of the geometry changes but the intensity of the profile is just linearly stretched. When we would have used a function like given in equation 4.12 to light the cushion the intensity of the border section would be higher as the slope of the geometry becomes less steep which means that the border intensity of the left border does not resemble the border intensity of the top border section of the cushion.

The implementation of the cushion rendering follows the four steps of construction from section 4.2.3 closely. The base square shape cushion is precalculated from equation 4.2 and stored in a one channel 2D luminance (or intensity) texture t_1. The stretched version of the cushion as described by equation 4.3 is implemented by stretching the texture t_1 over a rectangular shaped polygon. This step is actually not rendered in the implementation phase as the correct portions of the cushion are directly rendered in step two of the construction process.

Step two of the construction process is implemented by subdividing the construct shape in three sections as depicted in figure 4.1 (b). The three sections are rendered using rectangular polygons and textured using texture t_1 where the polygon coordinates are mapped to texture coordinates. The luminance values are now stored in the alpha channel of the color buffer.

The scaling functions described in equations 4.6 through 4.11 are also implemented as one channel 2D textures. Because scaling functions D, F, D' and F' are essentially rotated versions of the same scaling function we implement these scaling functions as one scaling texture t_2. Scaling functions E and E' are also rotated versions of the same scaling function which are implemented using a scaling texture t_3.

In step four of the construction process textures t_2 and t_3 are mapped to the polygons D, E, F, D', E' and F' of figure 4.4 (c) using the correct rotations by setting the corresponding texture coordinates. All these textures are rendered using multiplicative texture blending with the alpha value of the colour buffer. The ambient intensity value I_a is applied by using multiplicative blending. The luminance of the final cushion is now stored in the alpha channel of the colour buffer. A colour can now be rendered for the cushion by using multiplicative blending with the cushion color and the luminance alpha channel of the colour buffer.

The resulting rendering is depicted in figure 4.4 (d).

While rendering nested constructs we may replace geometry of parent constructs with the geometry of child constructs as explained in section 4.2.3. This allows us to render child constructs on top of parent constructs. This method is fast since we do not have to take any values from parent constructs into account.

The rendering process is efficient as it requires only a few texture operations for each cushion. Memory usage is also low as there are only three textures required for rendering all cushions. Furthermore, rendering nested constructs is done by simply rendering the construct tree in a top down fashion. This allows us to display many hundreds of cushions at once in an interactive system.

4.2.5 Border profiles

The geometry of the border of the cushion profile must be chosen such that the resulting visualization clearly depicts the nesting structure of the constructs. The border profile function is described by function g in equation 4.1. As the height value of the function H is directly mapped to the intensity value used for shading the cushion (see equation 4.13) the function g represents the geometry height as well as the cushion intensity value profile.

We experimented with various profile functions for g. We decided to use a parameterized cubic Bézier function for g. The cubic Bézier function for four control points $\mathbf{P}_i$ is defined in equation 4.14.

$$B(t) = \mathbf{P}_0(1-t)^3 + 3\mathbf{P}_1 t(1-t)^2 + 3\mathbf{P}_2 t^2(1-t) + \mathbf{P}_3 t^3 \qquad \text{for } t \in [0,1] \tag{4.14}$$

This Bézier curve is described as a parametric function. However the function g is described as an implicit function. This is not a problem as we can precalculate the Bézier curve at a desired quality such that y-values can be looked up given an x-value.

The four control points $\mathbf{P}_0, \ldots, \mathbf{P}_3$ can not be freely chosen. The function g is restricted by the the following properties:

$$g'(1) = 0,\ g(0) = 0 \text{ and } g(1) = 1$$

These properties ensure the shading is continuous and the border has height zero at the contour and height one at the end of the border section. The function of g is limited to the domain $[0,0] \times [1,1]$.

The control point $\mathbf{P}_0$ is set to $(0,0)$ as $g(0) = 0$ and the control point $\mathbf{P}_3$ is set to $(1,1)$ as $g(1) = 1$. This leaves control points $\mathbf{P}_1$ and $\mathbf{P}_2$. The y-coordinate of control point $\mathbf{P}_2$ must be set to 1 such that property $g'(1) = 0$ is ensured. The x-coordinate can be freely chosen from the domain $[0,1]$. When the x-coordinate is equal to one it is coinciding with $\mathbf{P}_3$ and gives a conflict with property $g'(1) = 0$. The result of this conflict is non continuous shading. Control point $\mathbf{P}_2$ is chosen such that the resulting curve is symmetric. The y-coordinate is set to 1 and the x-coordinate is chosen from $[0,1]$ depending on the value chosen for the y-coordinate of control point $\mathbf{P}_2$. Summarizing, the four control points are set to:

$$\left.\begin{array}{lcl} \mathbf{P}_0 & = & (0,0) \\ \mathbf{P}_1 & = & (0,j) \\ \mathbf{P}_2 & = & (1-j,1) \\ \mathbf{P}_3 & = & (1,1) \end{array}\right\} \quad \text{for } j \in [0,1]$$

The Bézier curve border profile function is depicted in figure 4.7. This figure also displays the four control points.

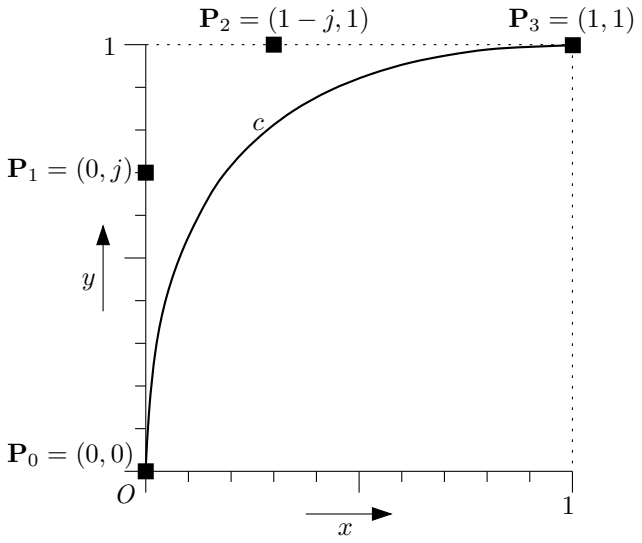


Figure 4.7: Cubic Bézier curve border profile function c with fixed control points $\mathbf{P}_0$ and $\mathbf{P}_3$ and control points $\mathbf{P}_1$ and $\mathbf{P}_2$ which can be positioned with parameter j to change the shape of the curvature.

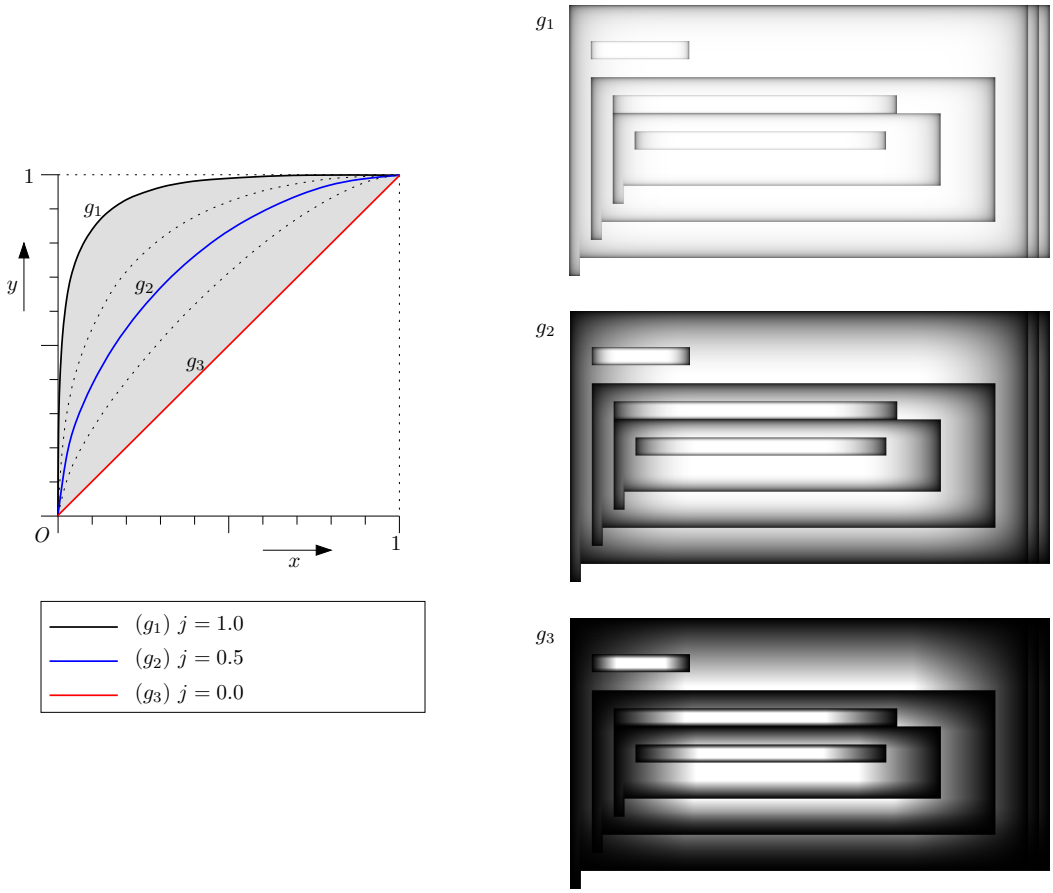
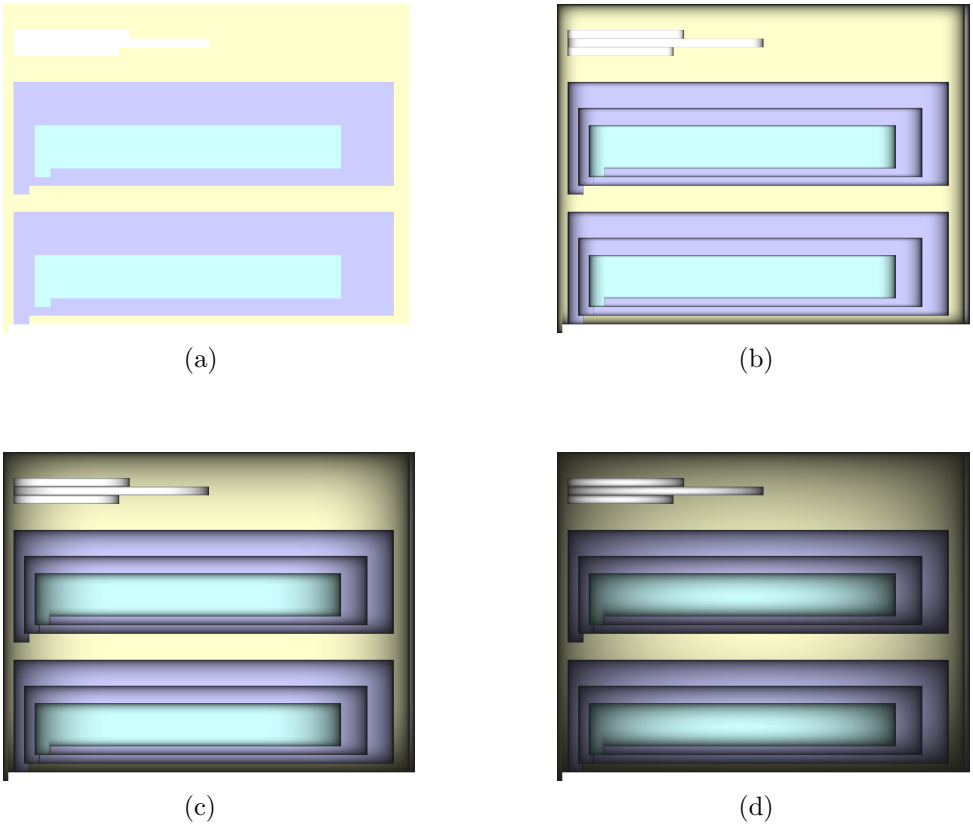


Figure 4.8: (a) Cubic Bézier curves used for cushion border profile for various values of j, parameter j controls the shape of the curvature, the shaded area displays the range of the curves; (b) Border profile functions applied to visualization

The border profile functions for various values of j are displayed in figure 4.8 (a). The functions from figure 4.8 (a) applied to the cushion visualizations are depicted in figure 4.8 (b). For low values of j (function g_3) there is almost no curvature and the light intensity increases almost linearly. This results in dark cushions where the border between various cushions is difficult to see. Also, the curvature near coordinate (1,1) is very small which results in a "banding" effect between the border shading of the cushions and the flat top of the cushions. For high values of j (function g_1) the curvature is very steep at the start of the function which means that the dark area of the border section is small. This results in cushions that are bright and have less pronounced borders. We concluded that values of j that lie near 0.5 give the best results (see function g_2). The shading is continuous without banding effects and the borders are clearly visible. The desired effect of cushion nesting by shading the cushions is best visible by using a value of $j \approxeq 0.5$ depending on the zoom level.

4.3 Layout

As programmers are familiar with text editors when editing and viewing source code we decided to keep the original text structure of the source code as they would appear in a regular text editor.



(a) (b)

(c) (d)

Figure 4.9: (a) Border width $w_b = 0$; (b) Border width $w_b = 0.06$; (c) Border width $w_b = 0.25$; (d) Border width $w_b = 0.50$

As is common we use a fixed width font to display the source code so the indentation of the code will be displayed correctly.

The syntactic constructs are displayed at the position of the text string that represents the syntactic construct. The connection between the construct visualization and the text string is made intuitively because they both occupy the same space. There is very little room for changing the layout of the constructs as these constructs are related to the text string. However, for the constructs that have body height h_B larger than one the width w of the construct can be expanded. This is useful in the nesting visualization as the margin surrounding the constructs give a better picture about how the constructs are nested. Expanding the constructs width to the left is usually not necessary as the programmer usually indents the constructs in such a way that a margin is already present. However increasing the width to the right of the construct does give better pictures as we create a margin that very likely may have been non-existent before the width increase. As we increase the width of a construct the width of the parent construct in the construct nesting order may also need to be increased to prevent child constructs from overlapping their parent constructs and to create a margin. The width expansion is visualized in figure 4.10. A construct with width w is increased in width to a construct of width w' by extending the left by m_l and the right by m_r. Note that the body height must be larger than zero as we may not expand the leading line to the left or the trailing line to the right as these areas may be occupied by other syntactic constructs.

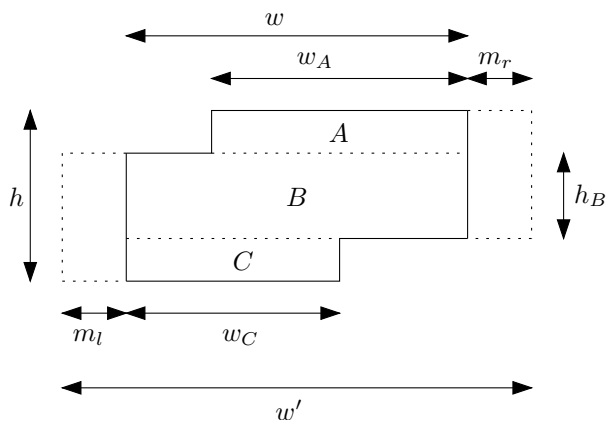


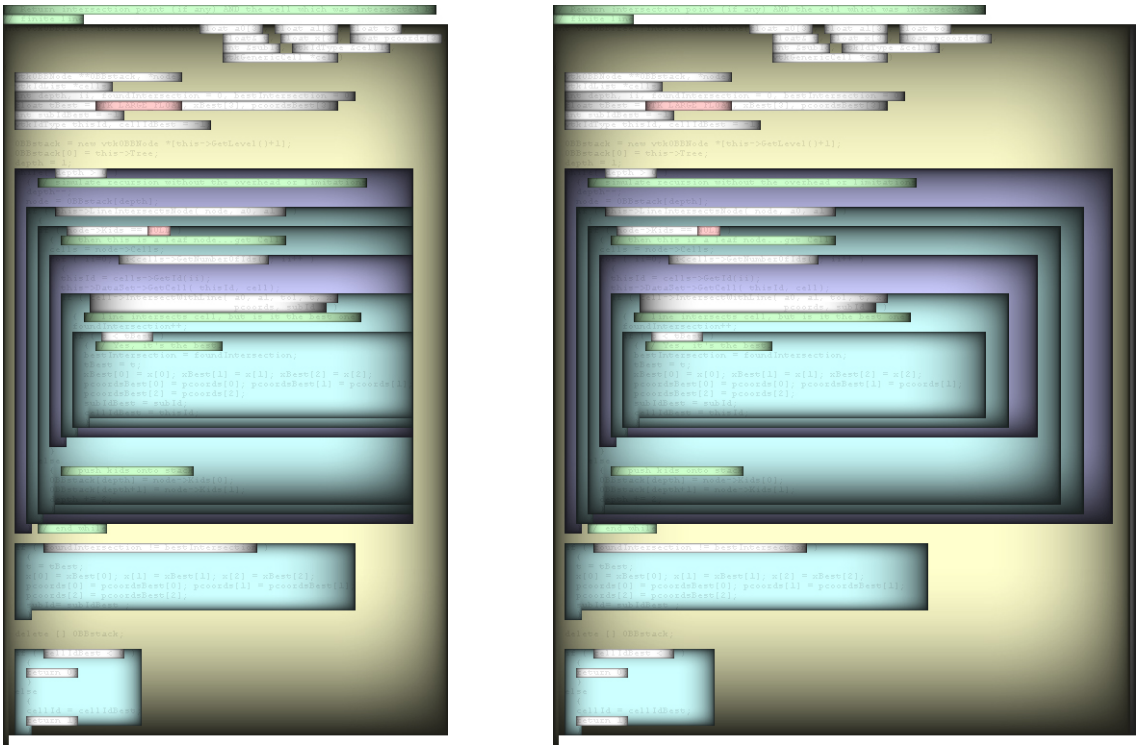
Figure 4.10: Increasing the width of a construct to the left with m_l and to the right with m_r.

Initially the width of the constructs is determined by the smallest possible contour surrounding the textual representation of the construct on the level of characters. The next step is to increase the width of constructs with body heights h_b greater than one that have no margin between the right edge and one of their children's right edge (the right edge position is equal or to the left of one of its child construct's right border). Before resizing the construct's width the algorithm recursively calls itself for every child construct of the construct that must be resized. Figure 4.11 (a) displays constructs that have not been expanded in width and figure 4.11 (b) shows the same constructs after width expansion.

4.4 Visibility

We visualize three types of data. The files, text and constructs. Visualizing all these types at once may not yield a very clear visualization. Depending on the required visualization the focus may be on the constructs rather than the text or vice versa.

It is possible to focus on the text or the constructs by changing the alpha transparency of the constructs (α_c) and text (α_t). The constructs are rendered on top of the background and the



(a) (b)

Figure 4.11: (a) Constructs without width expansion; (b) Constructs with width expansion to the right

text is rendered on top of the constructs. The alpha transparency can be changed by the user via sliders.

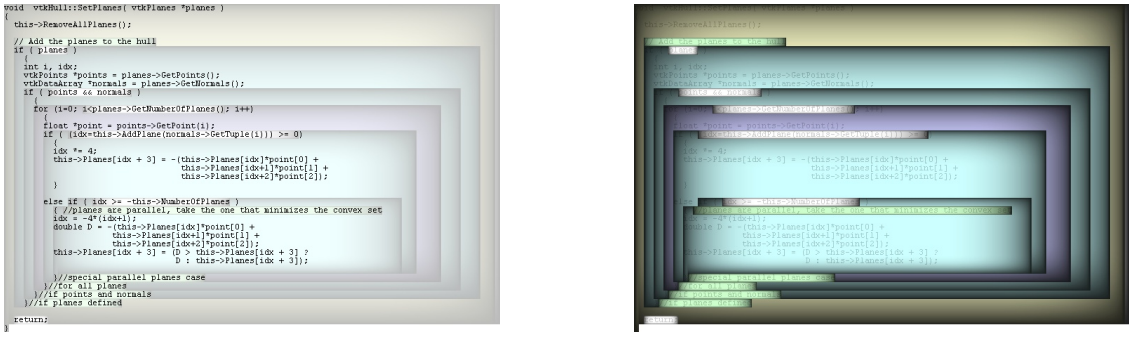
Figure 4.12(a) displays a rendering which focuses on the text with alpha transparencies set to $\alpha_c = 70\%$, $\alpha_t = 0\%$ and figure 4.12(b) displays a rendering which focusses on the constructs with alpha transparencies set to $\alpha_c = 0\%$, $\alpha_t = 80\%$.

Projects consists of multiple files. These files can be visualized in columns. However, it is common that only a certain set of available files is needed in the visualization. This is why it is possible to explicitly define which files to visualize. The files can be selected from a list or inclusion-order tree of available files.

4.5 Navigation

When visualizing large amounts of data it becomes very important to be able to navigate through all this data. We describe as before the various means of navigation for each of the data types *files*, *text* and *constructs*.

Files are displayed in vertical columns. These columns are visualized in a view port. The column width is the same for all columns as the average line width of all files is commonly almost the same and there would be no need in having columns of different widths. The column width can be changed linearly between a width such that all columns fit on the view port and a width such that only one column fits on the view port. This puts the focus either on a single file or on all files or anywhere in between if only a small set of files needs to be focussed.



(a) (b)

Figure 4.12: Cushion transpareny (α_c) and text transparency (α_t) can be modified. Transparencies are set to (a) $\alpha_c = 70\%, \alpha_t = 0\%$; (b) $\alpha_c = 0\%, \alpha_t = 80\%$

When the width of the columns is larger than the width of the view port it is possible to scroll the columns horizontally using a horizontal scrollbar. When the height of the contents of the columns is larger than the view port height the columns can be scrolled up or down by using a vertical scrollbar. The vertical scrollbar scrolls the contents of all columns up or down when possible. The range of the vertical scrollbar is determined by the height of the bounding box surrounding all the columns.

A typical usage scenario would be to display two points of interests in two different files (for example to compare two functions). These two points might not be located on the same file lines. Therefore it is possible to scroll the columns individually from each other using a mouse drag function or the keyboard. This enables the possibility to position multiple points of interest from multiple files next to each other.

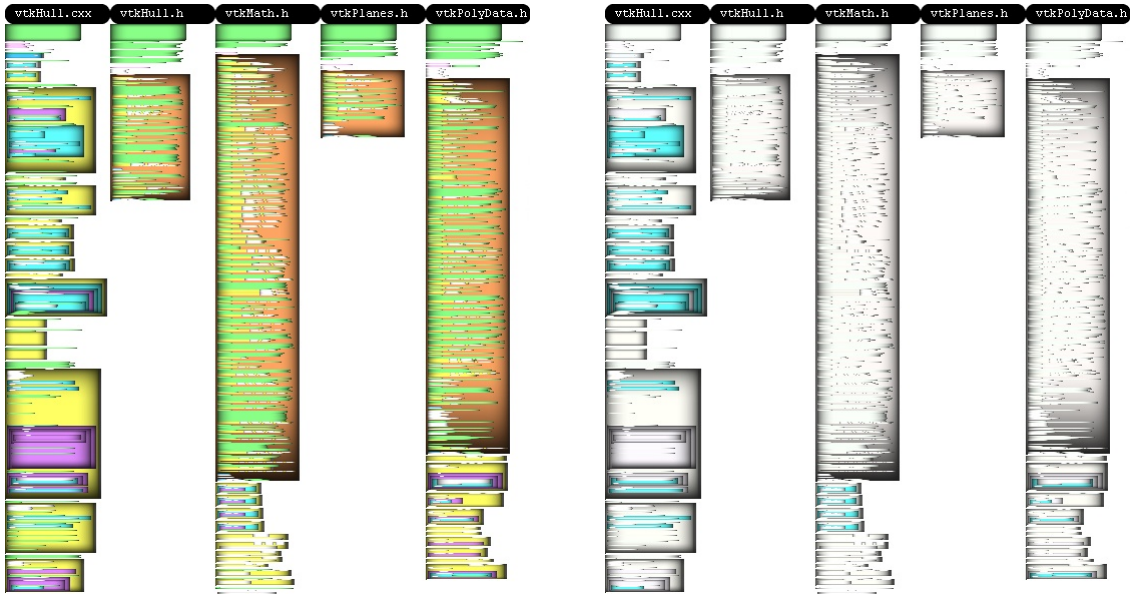
Navigating through the text is similar to navigating through the files using the scrolling functionality. It is similar to navigating through the text using a text editor. However, when the text is blended with the background or the background colour resembles the text colour it is not easy or even impossible to read the text. Therefore a mouse cursor mode is available which acts very much like a spotlight on the level of the source text. This cursor mode is called the *spotlight cursor*. The background surrounding a limited region $[0, w]$ x $[0, h]$ centred at the cursor position is changed to white and the text in this region is changed to black at the cursor position and gradually changes back to the background colour as the border of the region is reached according to some luminance function m. This means that there is a maximum contrast between the text and the background at the location of the mouse cursor which means the text is easily readable (provided the text font is not too small). The luminance function m we used for the blending is given in equation 4.16. This function has maximum intensity at the centre $(w/2, h/2)$ and slowly drops in intensity towards the borders. The gradient of this function is equal to zero at the borders which means that there is no banding effect when we use this function to blend the white colour with the background graphics.

$$l(x) = \cos(2\pi x - \pi) + 1 \qquad \text{for } x \in [0, 1] \tag{4.15}$$

$$m(x, y) = l(x/w)l(y/h) \qquad \text{for } x, y \in [0, w], [0, h] \tag{4.16}$$

The spotlight is implemented by storing the intensity values in a one channel 2D luminance texture. The black text and white colour are then blended with the background using multiplicative blending with the values from the texture. An example of the spotlight cursor is shown in figure 4.14 (a).

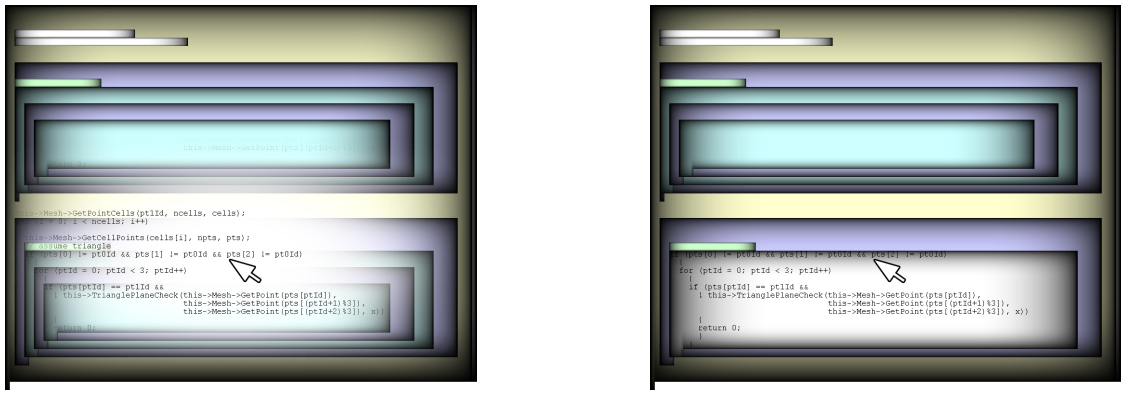
Navigation through the syntactic constructs is similar to navigating through text. However, it is very common that a distinction in the different syntactic construct types must be made. This



(a) (b)

Figure 4.13: Color desaturation (a) Before desaturation; (b) After desaturation

distinction can be made in the visualization by applying different colours to different types of constructs. Colour schemes can be saved and loaded into the application. This allows for saving colour profiles that focus on different aspects and it also allows to change the colour profile in order to change the focus to another area of interest. Operations on the colour scheme are also possible. A typical usage scenario would be that only one construct type must be highlighted temporarily. For this purpose it is possible to highlight a construct by means of desaturating the colours from all the other construct types. Completely desaturating the other colours removes the colouring scheme completely which might not always be what is wanted. Therefore the amount of colour desaturation can be controlled. Construct highlighting by desaturating the colours of other constructs is depicted in figure 4.13. The desaturation factor has been set to 95% in this picture and the focus was on the iteration statements which are coloured blue.



(a) (b)

Figure 4.14: (a) Spotlight cursor; (b) Syntactic cursor

Another typical usage scenario would be that the user is only interested in one particular construct

located below the cursor position. To this end a cursor mode is available that highlights the construct that is visualized below the current cursor position. The text in the highlighted construct is drawn in black with alpha value of zero (independent of the global text alpha value) and the cushion colour is drawn in white. This guarantees the maximum contrast between text and text background such that the source text can be easily read. Any deeper nested constructs that may be part of the highlighted cushion are not drawn so the main focus is on the construct below the cursor position. An example image is displayed in figure 4.14 (b).

It is possible to zoom the visualization. By zooming out the visualization it is possible to get an overview of the large amounts of data being visualized. Zooming is based on changing the font height of the text. This is because the cushion shape is determined by the font's width and height. It is possible to visualize many files together in a single view port. This is illustrated in figure 4.15. This picture displays a source file in the first column together with header files that have been included by this source file in the remaining columns. About 5000 lines of code are visualized.

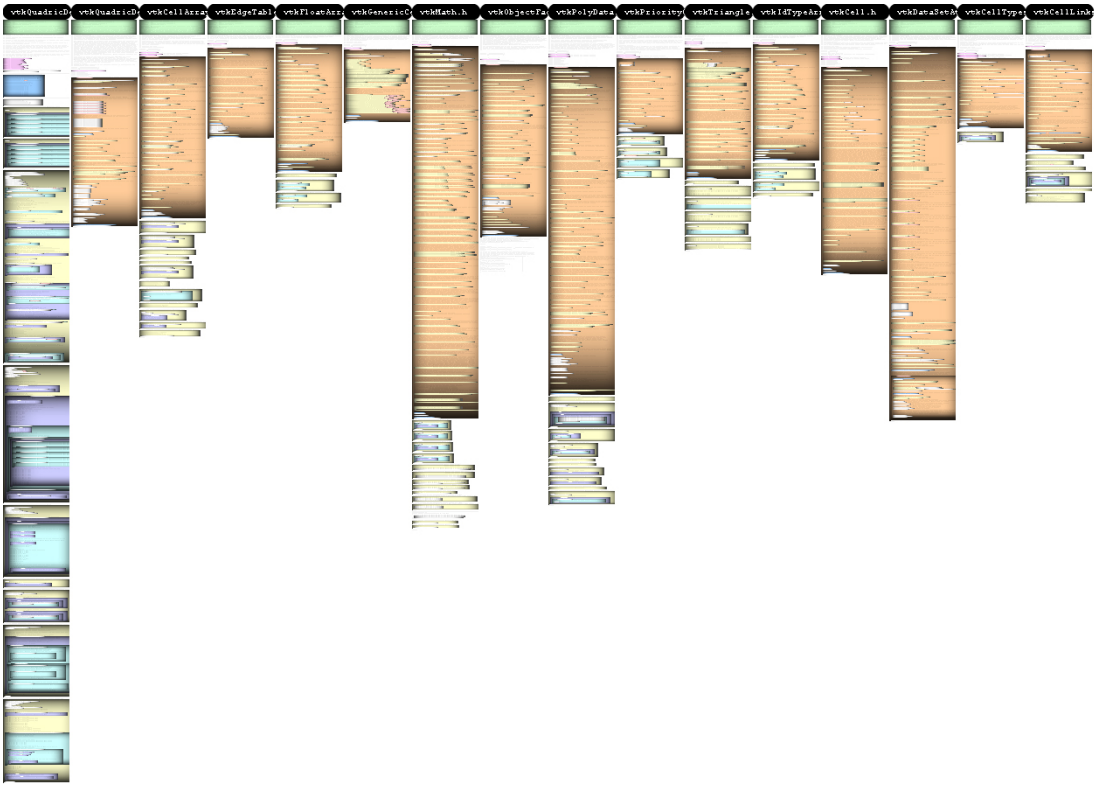


Figure 4.15: A visualization of 16 files showing over 5000 lines of code

The files and constructs in the visualization may be selected. This selection is visualized by colouring the selected items with a special selection colour. This selection can be used as the input to a query. The query can be used to navigate through the source code. This is explained in chapter 5.

Chapter 5

Queries

5.1 Introduction

The visualization as described in the previous chapter gives a global overview of the syntactic constructs present in source code and the way these constructs are nested. However, in order to better analyse and navigate the source code we need more information on the constructs. To this end we designed two query-models. The first model, model M_1, is designed to visualize query results on a global level. An example of such a query is: "what type do the constructs have?". In this example a colour is assigned to each construct in the visualization based on the type of the construct. This gives a quick and global overview of the construct types. The second model, model M_2, is designed for navigating and analysing subsets of constructs. Examples of such queries are: "where are the base-classes of this class declared?" or "where is this identifier declared?". The second model requires a selection method and a method of visualizing the result of the query.

The query-models are described in section 5.2. The design of the models is described in section 5.3.

5.2 Query models

A query in query-model M_1 can be formulated as a function Q_1 which takes n query specific arguments $a_1, a_2, \ldots, a_n$ and returns a construct color map M:

$$Q_1(a_1, a_2, \ldots, a_n) \rightarrow M$$

The colour map M maps each construct in the visualization to a colour. The number and type of arguments to this query are query-specific.

Query-model M_2 is based on construct selection. A query in this model can be formulated as a function Q_2 which takes a selection S as input and returns a query result. This query result consists of a selection S and a text string T:

$$Q_2(S) \rightarrow S \text{ x } T$$

The text string T in the query result is present to allow for detailed information about the query result which cannot be expressed by a construct selection. An example is information describing access rights to base classes or the number of selected elements.

File selections are also considered constructs so the selection S may contain file selections. This allows for queries that take a complete file as input.

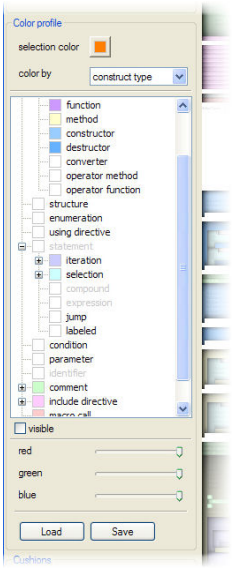


Figure 5.1: Controls to modify query settings. This example shows controls that can be used to modify the color map used in coloring constructs by contruct type.

Because the output of a query contains a selection it is possible to apply a query directly to a query result to form cascaded queries. If we have a query $q_2 \in Q_2$ that selects the base classes of a selected class it may for example be used to find the base class of the base class of a selected class by applying the same query q_2 to itself, i.e. $q_2(q_2(s))$. But it may also be a combination of different types of queries, like: "show the protected members (q_2') of the base class (q_2) of the selected class" which would look like $q_2'(q_2(s))$.

The queries from model M_1 and model M_2 have access to the complete fact base. This means that queries have access to the syntactic data as well as the semantic data.

5.3 Examples

Queries from query model M_1 can be selected from a list of available queries. The colour profile is updated according to the selected query. Query specific settings can be modified in the main user interface by using appropriate controls. An example of query specific controls is displayed in figure 5.1. Controls are displayed for a query that applies a colour to each unique construct type in the visualization. A tree control is available that can be used to modify the colours for each unique construct type.

Queries from query model M_2 can be selected by selecting cushions from the visualization followed by pressing the right mouse button. This action results in the display of a dynamic popup menu that lists all available queries that "accept" the current selection. A selection is accepted by a query if the selection contains constructs that can be handled by the query. The query acceptance can be formally described by a function F applied to a selection S which returns a boolean $\mathbb{B}$:

$$F(S) \rightarrow \mathbb{B}$$

An example of a query that accepts only a subset of the constructs is a query that, given a class selection, returns the public members of a class. This query only accepts constructs of the type class. The method of only displaying queries that can be applied to the current selection is important because the user immediately has an overview of the available queries. If on the other

hand we would have just given all available queries this might be overwhelming to the user and puts the task of determining which queries can be applied to the current selection at the user side. This allows the user to select a query that can not be applied to the current selection which not what we want.

(a) (b)

Figure 5.2: Query selection process; (a) Available queries are listed for current selection; (b) Result after selecting "base classes" query.

Figure 5.2 (a) shows the popup menu that is displayed after class `vtkCellLocator` has been selected and the right mouse has been pressed. The selection colour is red and classes are colour coded orange. The popup menu lists the available queries that accept class constructs. Figure 5.2 (b) displays the resulting selection after the "base classes" query has been selected from the popup menu. The result is also listed in the output text area below the visualization. The base class selected is the `vtkLocator` class. The resulting class construct represents the base class of the originally selected class construct. The base class is defined in a file that was not initially visualized (i.e. `vtkLocator.h`). If a query result contains constructs that are located in files that are not visualized these files are automatically opened and visualized.

Chapter 6

Scenarios

The scenarios described in this chapter use sample code from the Visualization ToolKit (VTK) [12]. All scenarios use the source file "vtkOBBTree.cxx" from VTK and all of the first level include files included by this file.

In the following we sketch a number of user-level questions and try to show how they are approached by our visualization tool. We validated the various results obtained with our visualization tool by manual inspection of the source code text.

6.1 Scenario A

"How are comment constructs and high-level constructs such as classes and subroutines distributed?".

This can be answered by selecting a query that colours constructs by construct type. Comment constructs and high-level constructs such as classes and subroutines are set to visible and lower level constructs such as identifiers are set to invisible. The resulting visualization is displayed in figure 6.1.

The green constructs represent comments. All files have a comment of the same size as their first construct. This construct is most likely a standard comment header with copyright information and file version information. The orange constructs depict class constructs. It appears that the convention is to have about one class per include file. Class member method declarations are coloured yellow. A number of class declaration constructs are followed by member declarations. These declarations are inline functions. It looks like line comment, which is coloured green, precedes every class member method declaration. A closer inspection on the class member declarations by zooming in and using the spotlight cursor as shown in figure 6.2 confirms this.

We can conclude that the distribution of constructs is influenced by applied coding standards.

6.2 Scenario B

"Are there any macro constructs used, and if so, how are these constructs distributed over the files?".

This can be answered by choosing a colour profile such that macro calls are colour coded red and high-level constructs are colour coded white. Low-level constructs are not visualized as these are not important in answering this scenario. The colour coding ensures that the macro invocations are clearly visible. To make the contrast between the macro calls and the background even more

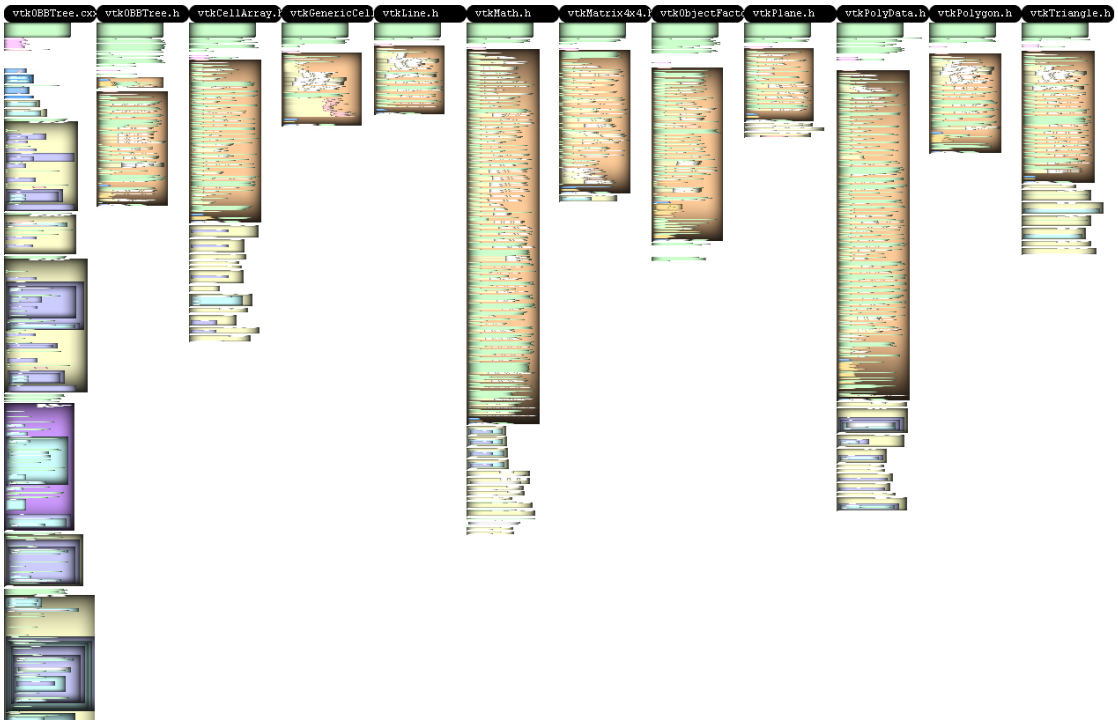


Figure 6.1: A visualization of a set of source files using a color scheme that applies a unique color for each unique construct. Comment and all the high level constructs such as classes and subroutines are visible.

clear we increase the ambient light factor to reduce the shading effect of the underlying cushions. The resulting visualization is displayed in figure 6.3.

A first observation shows that there are macro calls in the header of every class. A closer inspection shows that the macros in the header (`VTK_GRAPHICS_EXPORT, VTK_COMMON_EXPORT` etc.) export classes. This is an indication that the classes are part of a class library. Another macro that is present in the header of every class is the `vtkTypeRevisionMacro` which sets type information for every class. We can see that other macros are used throughout the code but that these macros do not appear according to a pattern.

We can conclude that macros used in classes are placed according to a coding standard and that the classes that are visualized are part of a class library. Furthermore, macros are used throughout the rest of the code but not according to a pattern.

6.3 Scenario C

"What are the deepest nested constructs with respect to iteration statements and selection statements?"

The default shading used in visualizing cushions can be used to determine the nesting depth. However it is difficult to quickly see where the deepest nested construct are located. In order to more clearly visualize the nesting depth we use a colour scheme that applies a colour, from a gradient colour map, to each construct depending on the nesting depth. The colour gradient starts with the colour blue, followed by cyan and yellow, and ends with the color red. The deepest nested construct is colour coded red and the top level constructs are colour coded blue. Only iteration statements, selection statements, subroutines and classes are visualized. The subroutines

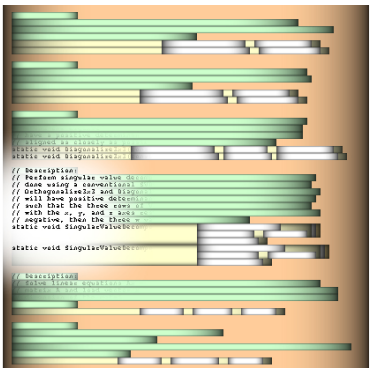


Figure 6.2: Zoomed in section of the visualization from figure 6.1 with the spotlight cursor mode enabled revealing line comment preceeding every class member declaration.

and classes are visualized in order to get an impression of the global structure. The resulting visualization is displayed in figure 6.4.

The include files are all completely visualized. The source file (`vtkOBBTree.cxx`) is not visualized entirely on a single screen. This is because the file contains too many lines (˜1900) to be displayed at once. Scrolling through the file solves this problem. An observation can be made that the deepest nested constructs are all located in the source file. This is expected as the source file contains the implementation of the class methods. The deepest nested constructs are quickly found as these constructs are colour coded bright red. The constructs are part of the `intersectWithLine` method and have a nesting depth of 7.

We can conclude that the locations of the deepest nested construct can be quickly found by using a colour scheme that colours constructs according to their nesting depth.

6.4 Scenario D

"What is the complexity of the classes in the files with respect to the number of base classes, what class has the longest base class chain?"

This scenario can be answered by using a colour scheme that applies a colour depending on the maximum length of the base class chain(s). We use such a colour scheme. Classes that have no base class (base class chain of length zero) are colour coded yellow. Classes that have the longest base class chain are colour coded red. Classes with base class chain lengths between the maximum and minimum are assigned a linearly interpolated colour. The resulting visualization is displayed in figure 6.5.

The class complexity with respect to the length of the base class chain is directly mapped to the colour of the class constructs. The class construct with the longest base class chain is directly spotted as class `vtkPolyData`. It can be observed that the file `vtkOBBTree.h` contains a class that has no base classes. A closer inspection reveals that this class is the class `vtkOBBNode` which represents a node in the `vtkOBBTree` datastructure.

We can conclude that the query used in colouring the constructs can answer the question of the scenario very well. In general the queries that colour constructs can be added as plug-ins to the system and can be used to answer other questions about the code that can not be answered with the default set of available queries.

Figure 6.3: A visualization of a set of source files using a color scheme that applies a red color to macro constructs and a white color to all other visible constructs. The ambient light intensity is set to 0.8 such that the cushion shading is less dominant and the macro call constructs are more clearly visible.

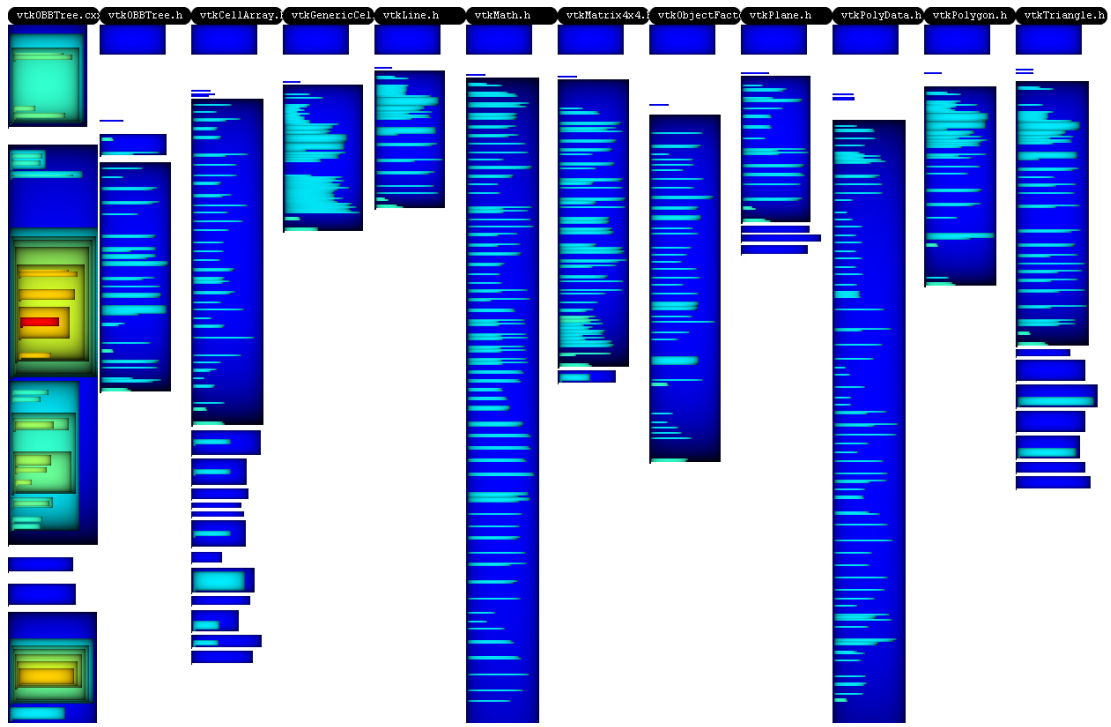


Figure 6.4: A visualization of a set of source files using a color scheme that applies a gradient color map to each construct depending on the nesting depth.

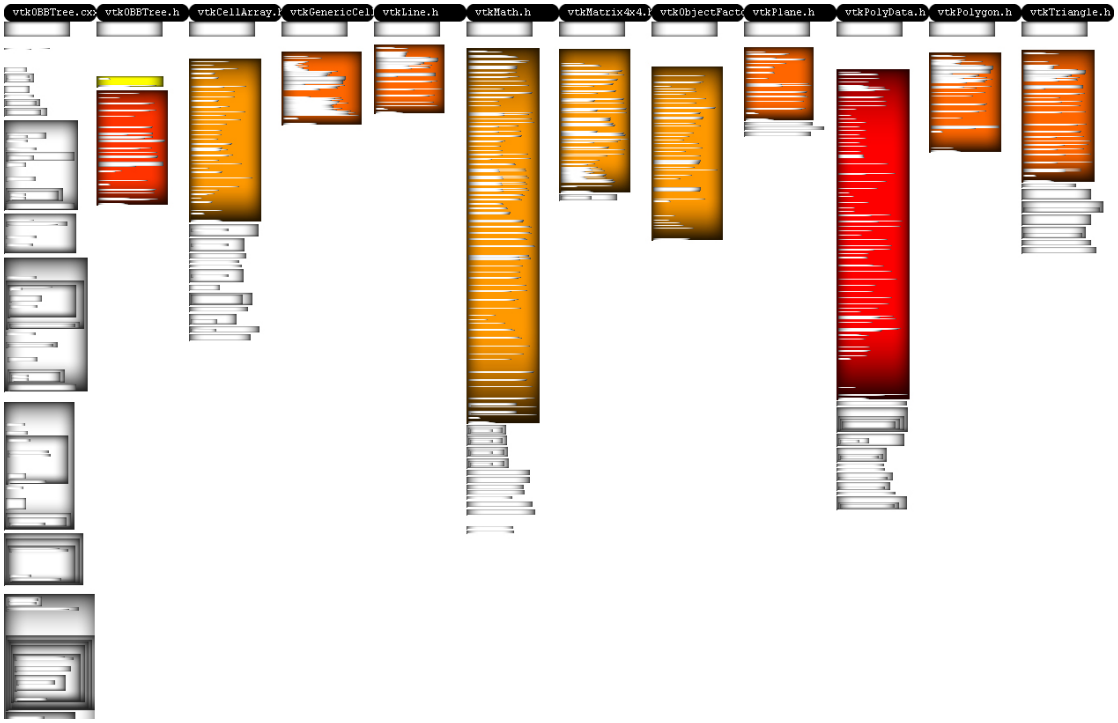


Figure 6.5: A visualization of a set of source files using a colour scheme that applies a linearly interpolated colour to each class construct depending on the maximum path length to the top-level base class.

Chapter 7

Conclusions

Our main objective was to design and implement a code structure visualization tool that helps programmers understand and manage source code. This has been accomplished by visualizing the structure of source code and by providing interactive navigation and query systems.

An important part of the code structure visualization program is the fact extractor. This program is needed to extract structural facts from source code. We listed some requirements for the fact extractor in section 2.3. We will list these requirements again with a short summary of how each requirement is realised.

1. fact extraction of C/C++ source code

   The GCC C++ compiler forms the basis for the fact extractor. This compiler contains a C++ parser which is extended for the purpose of fact extraction. This means that pure C++ source code is supported. However, pure C code is also treated as C++ code. This means that C code is only supported when it is valid C++ code. There are a few C specific cases that are not also part of the C++ standard and will produce errors. The differences between ISO C and C++ are described in Annex C of the C++ standard [8]. However, we do not regard this as a problem as the differences are small and most of the C code complies with the C++ standard.

2. fact extraction of any C++ based project

   The goal was to support any C++ based project. This means that the fact extractor should be able to extract facts regardless of the target compiler of the project. However different target compilers have small differences which makes it impossible to write a fact extractor that supports any target compiler as the differences are not known in advance. It is possible however to simulate target compilers using the fact extractor. This process is described in section 3.7. Using a method similar to the method described in this section support for simulating other target compilers may be added. The compilers supported by the fact extractor are: GCC C++ 3.4.x, Microsoft Visual C++ 6.0, 7.1 and 8.0 (beta). Most of the major compilers are supported and other compiler support can be added to the fact extractor. This means that, in theory, any C++ based project can be supported. However, due to non-strict compliance to the C++ standard it might be possible that user code must be patched in order to comply with the standard. This is a problem inherent to the used parser.

3. complete syntactic and semantic fact extraction

   The goal for the fact extractor was to be able to extract a complete syntactic fact base. This has been realised by extending the parser's recursive descent functions such that every construct is extracted. However, some constructs are not extracted by the fact extractor

as these constructs were not important for our visualization purposes. By extracting these facts the fact base would become very large. However, by extending the remaining recursive descent functions it is possible to extract all syntactic facts. Support for new syntactic constructs can be iteratively added.

Semantic facts are extracted for the extracted syntactic facts. Since we do not extract the complete set of syntactic constructs we do not extract the complete set of semantic facts.

4. mapping of syntactic constructs to source code

   For our visualization purposes we needed detailed location information for every syntactic construct extracted by the fact extractor. This has been accomplished by extending the preprocessor's lexical scanner and the interface between the preprocessor and the parser. When extracting the syntactic constructs during the parsing process we couple the detailed location information with the constructs by extracting them from the preprocessor generated tokens which represent the construct in question. This process is described in detail in section 3.4.1.

5. platform independent

   In principle the fact extractor can be compiled on any platform supported by GCC. A list of supported platforms is provided on the website of GCC [5]. Support for the most common platforms GNU/Linux and Microsoft Windows is available. The fact extractor can be compiled on any linux platform without problems. The windows platform is supported by compiling the fact extractor with Cygwin [4]. Cygwin is a Linux like environment for windows. It provides a Linux API emulation layer through a DLL file.

6. well structured

   The goal was to provide a data structure optimized for visualization clients. This has been accomplished by providing a simple base data structure. The facts are ordered in a tree structure by their common nesting structure. Semantic facts are added to this tree structure as attributes to the tree nodes. The facts are saved as an XML formatted file. The XML structure is built up in the same way as the data structure. This data structure is very general and simple and can also be used in other tools not specifically written for code visualization.

7. efficient

   The fact extractor is as efficient as the GCC C++ parser plus a small overhead for the data structure construction and added postprocessing stage. The postprocessing stage and data structure construction phase may be improved with respect to efficiency, however, since the fact extractor is based on the C++ parser it can never be more efficient than the parser itself. In practice the speed is acceptable for our purposes.

Almost all requirements set out for the fact extractor have been met. The only requirement not completely met is that the fact extractor should be able to extract a complete set of syntactic and semantic facts. A small set of syntactic facts are not extracted by the fact extractor. The decision to not extract the complete fact base has been made deliberately as we did not need a complete fact base for our visualization purposes and a complete fact base would be unnecessarily large.

The fact extractor has been run on large projects such as the Visualization ToolKit (VTK) [12] with success. This project consists of over 700 classes and over 350,000 lines of code.

The visualization tool is meant as an aid for programmers. The main objectives set out for the tool are enumerated below. Each objective contains a small summary of how the objective has been accomplished.

1. provide a visualization that clearly depicts the structure of the source code displayed

In order to meet this objective we must first define what exactly makes up the structure of source code. Source code must be written according to some grammar. This grammar forces the source code to be structured. The grammar decomposes the source code into syntactic constructs. These constructs may be nested. The way these constructs may be nested is described by the production rules in the grammar. The nested syntactic constructs form the bases for our visualization tool.

As the visualization tool is meant as an aid for programmers we decided to display the syntactic constructs on top of the normal textual layout. Programmers are already familiar with the textual layout of the source code and will therefore easily recognize the structure.

We decided to graphically display the syntactic constructs as flattened cushions. This clearly depicts the border between various constructs but also depicts the nesting. Deeper nested constructs appear to be displayed "higher" due to the shading used in drawing the constructs. This gives a quick overview of the nesting structure.

We "enhance" the textual representation with flattened cushions. The text and cushions can be displayed simultaneously. However, depending on the situation the focus may lie more on the cushions than on the text or vice versa. This is why the cushions can be alpha blended with the background and the text may be alpha blended with the cushions and the background. In this way our tool can be turned from a classical text editor to a true syntax-highlighted editor by the simple move of one or two sliders.

2. provide means of displaying large quantities of code on a single display

   Large quantities of code can be displayed by displaying multiple files in column mode. The graphical representation can be zoomed out to allow for displaying more lines of code on a single display. This allows for displaying thousands of lines of code on a single display. An illustration is given in figure 4.15.

3. provide means of interactively navigating source code

   We defined navigation on a textual, syntactic and semantic level. The tool provides a means of scrolling through the text and constructs in a similar way of scrolling through files in a multi document interface. Furthermore the mouse cursor may be used to put the focus on a specific area of interest. Mouse cursor modes exists for focussing an area of interest on a textual basis, the spotlight cursor, and on a syntactic basis, the syntactic focus cursor. Navigation on a semantic level is provided via interactive queries.

4. provide an extendible query system that supports interactively querying source code

   For this objective we must define what the input to the queries is, what the output of the queries is and how we provide means of selecting input to queries and means of visualizing the output of the queries.

   We defined two query models. The first query model contains queries that apply a colour scheme to the constructs being visualized. The colour applied depends on the specific query and the input to the query. The input also depends on the specific query. An example of a query from this model is a query which applies a colour to a construct depending on the type of the construct. The input to this query is a mapping between the construct type and a colour.

   The second query model is based on selection. The input to queries in this model is a user defined selection of constructs and the output is another selection and a text string. This query model is ideal for navigating source code. The queries can be interactively selected by use of the mouse cursor. Example queries in this model are: a query that selects the base classes of a selected class construct or a query that selects the declaration of a given identifier. The query also returns an optional text string which is displayed under the visualization view port. This text string can be used to return a description of the result of the query that can not be expressed with a selection of constructs.

   Both query models can be extended with user specific queries through the use of plug-ins.

All objectives for the visualization tool have been met reasonably well. There is still some trouble with the queries as the semantic information is not complete. This is because the extraction of the semantic facts still has some problems. The main objective of the tool to be an aid for the programmer to better understand and manage source code has been met by providing the solutions to the listed objectives. The interactive query system in particular is very powerful as this query system may be extended with new queries by the user of the tool. This allows the user to enhance the navigation and the visualization depending on the specific user's needs and may serve to better understand and manage source code.

7.1 Future work

The visualization tool visualizes the structure of source code. However the source code can not be directly manipulated, since the fact extractor used by the visualization tool can not parse erroneous source code. A possible extension would be to add a fact extractor that can extract information from erroneous code and integrate the visualization with a code editor. This allows for visualizing the structure of code while editing the code.

The flattened cushions are shaded by directly mapping the height of the cushion profile to the light intensity. This method is very simple and allows for fast implementations. However the disadvantage is that no "real" lighting can be used. A possible extension is to use the geometry of the cushions for shading the cushions. This allows for changing the position of the light source. This may give a better visualization of the nesting structure in some situations.

The visualization tool can only be used in combination with C++ based projects. This is a clear restriction of the tool. It might be possible to extend the available fact extractors with extractors that can extract facts from other languages as well. It might also be possible that other tools are written that use the fact extractor presented in this thesis. To provide a common interface between the different tools (clients) and the different fact extractors we may introduce a middleware layer. This is depicted in figure 7.1. The middleware layer is responsible for providing a common API to the clients.

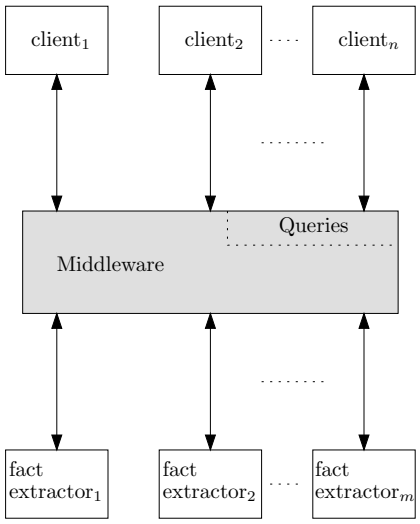


Figure 7.1: Middleware layer

It is conceivable that third party fact extractors may be used. This results in fact extractors which produce different data structures. The middleware layer is responsible for translating these different data structures into one common data structure that is presented through the middleware API to the clients.

Queries can be performed on the data structure offered by the middleware layer. It is possible that there are some very common queries that are used in a number of clients. To prevent the replication of the queries in different clients the middleware layer may offer a set of queries. Each different language may contain language dependent queries. The middleware layer may be used to request a list of queries that can be used in combination with the fact base. The queries from query model M_2 presented in chapter 5 are very suitable to serve as general queries. These queries may be represented by the middleware layer as C++ specific queries.

The middleware layer also offers the possibility of using multiple fact extractors that are able to parse a specific language. This may be useful if the fact extractors that are available are not complete and the user wants the information extracted from multiple fact extractors. The middleware layer may merge the facts generated by multiple fact extractors.

Appendix A

CSV User manual

A.1 Introduction

The Code Structure Visualization tool (CSV) is a tool for visualizing the syntactic structure of C++ source code and which aids in navigating source code.

The goal of the Code Structure Visualization tool is to get more insight in the structure of C++ source code. The tool visualizes source files in a programmer familiar layout (multiple document interface like layout). The source files are "enriched" with graphical cues about the structure of the source code. Interactive mechanisms are present which aid in navigating the source code.

A.2 Getting started

A.2.1 Installation

The Microsoft Windows version of CSV can be installed using a setup program. This program presents a wizard which will guide you through the installation procedure. The installer is created using the open-source Nullsoft Scriptable Installer (NSIS) system. This installer presents a setup interface commonly used in Microsoft Windows systems. It also creates an uninstaller which automatically removes the application.

A.2.2 Running

The installation wizard has created shortcuts in the start menu. These shortcuts can be used to start the application. A shortcut to the main application is available as well as shortcuts to load sample visualizations.

The process of opening already existing visualization files is done by selecting `File -> Open -> Visualization` from the file menu followed by selecting the file to be opened.

The process of creating a visualization from source code is by selecting `File -> Open -> Source` from the file menu followed by selecting a C++ based source file and entering project related settings in the project dialog when required.

Command line options

There are a number of optional command line arguments available. Following is a list of the command line options:

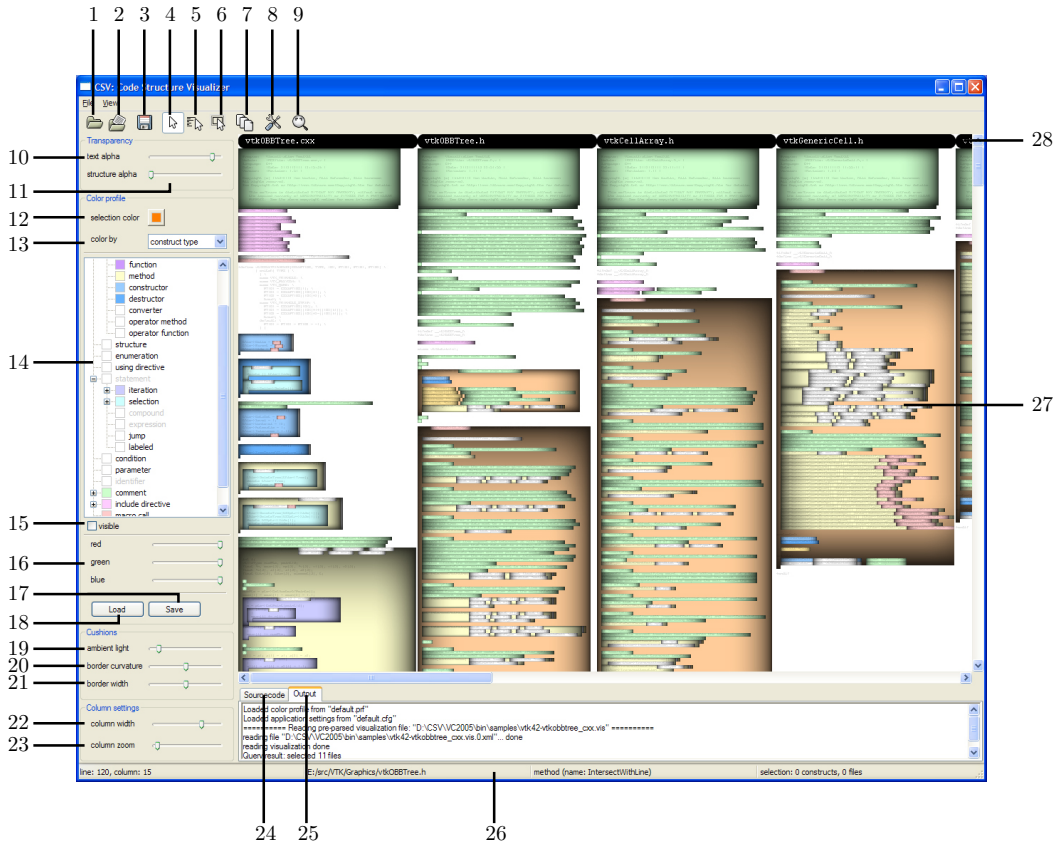


Figure A.1: Main user interface

Option	Description
-visualization <file>	Loads preparsed visualization on start up
-color-profile <file>	Loads custom color profile on start up
-settings <file>	Loads custom settings on start up

A.3 Functionality

A.3.1 Main User Interface

All common application interaction is through the main user interface. Controls for less common interaction or controls that are not suitable for display in the main interface are located in dialog windows. The main user interface of CSV is depicted in figure A.1. A description of the controls displayed in this figure are listed in table A.1.

Toolbar buttons	
(1)	Open visualization file
(2)	Open source file for fact extraction and visualization
(3)	Save current visualization to file
(4)	Normal cursor mode
(5)	Spotlight cursor mode

continued on next page

	continued from previous page
(6)	Syntactic focus cursor mode
(7)	Open file list dialog
(8)	Open settings dialog
(9)	Reset visualization to default settings
Visualization controls	
(10)	Text alpha value control
(11)	Construct alpha value control
(12)	Selection colour control
(13)	Colouring method control
(14)	Colour profile tree for available syntactic constructs
(15)	Toggles visibility of current syntactic construct
(16)	Colour controls for red, green and blue intensity of current syntactic construct
(17)	Saves colour profile to file
(18)	Loads colour profile from file
(19)	Ambient light intensity control
(20)	Border profile curvature control
(21)	Border width control
(22)	Column width control
(23)	Column zoom control
Diagnostics	
(24)	Preview of source code located under mouse cursor text area
(25)	Application diagnostics and query result feedback text area
(26)	Status bar displays information about cursor location
Visualization window	
(27)	Visualization view port which displays files in column mode
(28)	Column headers which display filenames

Table A.1: Main user interface components

The main user interface's controls listed in table A.1 are logically grouped. Functionality of the controls in each group is explained in the next sections.

Toolbar buttons

The toolbar buttons provide quick access to common tasks. Button (1) is used to open a visualization file. A visualization file is a combination of facts describing certain source code and the source code itself. After opening the file a graphical representation is displayed in the main window. Button (2) is used to open source files. After opening a source file the fact extractor will parse this file and the resulting facts are used to give a graphical representation of the structure of the source file. Before the fact extractor runs a context dialog window is displayed in which project information related to the opened source file can be entered. This is information such as include directories and macro definitions. The dialog is displayed in figure A.3 (left).

It is possible to save the visualization to disc. This is done by pressing button (3). It saves the current visualization which consists of facts describing the source code displayed and the source code itself. The visualization can consist of an arbitrary amount of preparsed source files.

The function of the mouse cursor when moved over the visualization window can be changed by selecting one of the push buttons (4), (5) or (6). These push buttons are mutually exclusive. Button (4) selects the normal mouse cursor mode, button (5) selects the spotlight mouse cursor mode and button (6) selects the syntactic zoom mouse cursor mode. When the spotlight mouse cursor mode is selected the text surrounding the mouse cursor's position will be highlighted. This cursor mode is used to navigate source code on a textual basis. When the syntactic mouse cursor

mode is selected the syntactic construct (cushion) located under the mouse cursor is highlighted. This cursor mode is used to navigate the code on a syntactic basis.

Usually source files include other files. These *include files* are not opened by default by the application. This is because there can be many include files which may be of no interest to the user. A tree or list of available include files can be displayed in an include file dialog window by pressing button (7). The include file dialog is displayed in figure A.3 (center).

There are many settings that are not suitable to be displayed in the main user interface. These settings can be displayed and manipulated in the settings dialog window which is accessible through button (8). The dialog is displayed in figure A.3 (right).

It is possible that by zooming and scrolling the overview of the current visualization is lost. A default set of visualization settings can be restored by pressing button (9).

Visualization controls

The visualization controls are located to the left of the visualization. These controls can be used to customize the visualization.

The alpha transparency of the source code text and the syntactic construct visualization can be changed such that the text or constructs are more or less prominent. The alpha values of the text and constructs can be changed by the slider controls (10) and (11) respectively.

The colour of selected cushions can be changed by using control (12). The method used in assigning colours to cushions can be changed with control (13). This also changes the subsequent controls depending on the chosen type of colouring method.

It is possible to map a syntactic construct type to a colour. This is useful when the focus is on certain kind of constructs. A list of available types with a preview of the current colour is displayed in the colour profile tree (14). The visibility and colour of the constructs in this tree can be changed with controls (15) and (16) which act on the currently selected construct. Colour profiles can be saved to file and loaded from file with controls (17) and (18) respectively which allows for creating different colour profiles for different purposes.

The graphical representation of the syntactic constructs can be changed. It is possible to change the ambient light used when drawing the constructs. This may be useful to light otherwise dark areas of certain constructs. Changing the ambient light can be done with slider control (19). The curvature of the border profile of the cushions can be changed with slider control (20). This can be useful in order to better display the nesting structure of the constructs. The border width can be changed with control (21).

To be able to display more lines of code on a single viewport the visualization can be zoomed. Also, the column width can be modified in order to display multiple columns on a single view port. The column width can be manipulated with slider control (22). The column width value lies between a width such that all opened files fit on one screen next to each other and a width such that only one column fits on the screen. Zooming is done with slider control (23). Zooming is done on character height. The lowest zoom value is one, such that every line of code is only one pixel in height, and the highest zoom value is a given maximum.

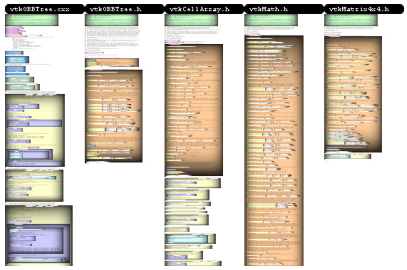


Figure A.2: Zoomed out to maximum

Diagnostics

When navigating a visualization it is common that the user wants to read the source code at a given mouse cursor position. However, depending on the visualization settings the source code may not be readable. This happens when the visualization is zoomed out too far or when the alpha transparency of the text is set to a value such that the text is not readable anymore. In those cases the user does not want to change the current visualization settings in order to be able to read the source code. To this end a text pane is present (24) underneath the visualization view port which displays the source code in a limited area around the current mouse cursor position. The size of the text in this pane is constant. This allows for reading the source code without having to change the visualization settings.

The application gives feedback to the user in the form of a text string. The feedback text is presented in a text area located underneath the visualization view port (25). The information can for example be information about the reading progress when opening a visualization file or diagnostic information from the parser when the fact extractor fails. This information is not displayed in a popup-window to allow for a non-blocking application. Information on query results is also displayed in this text area.

When moving the mouse over the visualization view port information on the current position of the mouse cursor in the file located underneath the cursor is presented in the status bar (26) of the application. The information is information on the line and column position of the cursor, the filename of the column, the construct type of the construct located at the cursor position and a summary of the currently selected constructs.

Visualization window

The visualization view port (27) displays source files in column mode. These files are enriched with graphical representations of the structure of the code. The files can be scrolled using the view ports horizontal and vertical scrollbar. The horizontal scrollbar scrolls all the columns left or right and the vertical scrollbar scrolls the contents of all the columns up or down when possible. Each column has a column header (28) located on top of the column. This header displays the name of the source file that is displayed in this column.

A.3.2 Dialogs

This section describes the functionality of the various dialogs which are available in the application. All the dialogs are accessible through the main user interface (see section A.3.1). Figure A.3 depicts all dialogs and table A.2 lists all the dialogs' components grouped by dialog.

Source file context dialog	
(29)	Load context from file
(30)	Save context to file
(31)	List of include directories
(32)	Adds a directory to include directories
(33)	Removes selected directory from include directories
(34)	List of macro definitions
(35)	Add macro definition
(36)	Remove selected macro definition
(37)	Additional compiler command line arguments

continued on next page

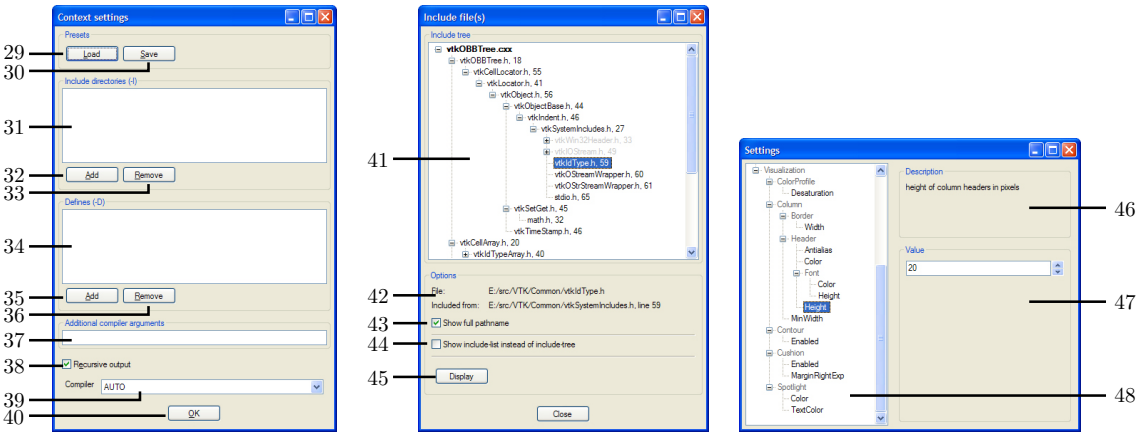


Figure A.3: Source file context dialog (left); Include file dialog (centre); Settings dialog (right)

	continued from previous page
(38)	Toggles recursive parsing
(39)	Target compiler simulation type
(40)	Apply context
Include file dialog	
(41)	List or tree of include files
(42)	Information on selected file
(43)	Toggle full path name in information area
(44)	Toggle between list view and tree view
(45)	Displays selected file graphically in main user interface
Settings dialog	
(46)	Description of selected setting
(47)	Area with dynamic controls for manipulating selected setting
(48)	Tree view of available settings

Table A.2: Dialog components

Source file context dialog

When opening a source file for visualization this file must be parsed first in order to extract facts from the source code that are needed for the visualization. However, this parser must run in a certain context. This context depends on the source project the opened source file belongs to. The source file context dialog is displayed when opening a source file which allows setting up the correct context settings for the source file. The dialog is displayed in figure A.3 (left).

The most important project settings are include directories and macro definitions. Include directories are displayed in a list box (31). The directories can be added to or removed from this list using buttons (32) and (33) respectively. Macro definitions are displayed in the macro definitions list box (34) and can be added to or removed from this list using buttons (35) and (36) respectively.

If the context can not be described by the include directories and macro definitions additional context parameters can be supplied in the additional parameter text field (37). Valid parameters are regular GCC C++ compiler command line arguments. This is because the fact extractor is based on the GCC C++ compiler.

The fact extractor extracts facts for the selected source file and any file that is included from this source file recursively. If the user is only interested in the facts describing the selected source file and not files included from this file the recursive checkbox (38) can be unchecked. This option

reduces required disc space significantly when saving the visualization. However, include files can not be opened as no facts are available for those files.

It is possible to select the target compiler for the selected source file in the target compiler dropdown list (39). The default value for the target compiler is `AUTO` which results in the fact extractor searching for a supported compiler on the current system. However, when there are multiple supported compilers available on the system or the automatic search fails it is possible to manually select the target compiler. The target compiler will be simulated by the fact extractor.

The context settings can be loaded from file or saved to file using buttons (29) and (30) respectively.

By pressing the OK button (40) the context settings are used and the fact extractor is started. If the fact extractor succeeds the source file is visualized in the main user interface.

Include file dialog

Source files usually include other files recursively. This means that a single source file may contain a tree of include files. An overview of these include files is displayed in the include file dialog which is displayed in figure A.3 (centre). The files can be displayed as an include tree or as a regular list of files. The list view is particularly useful when a quick overview of included files is needed without the need to see the include nesting structure. The files are displayed in (41). Switching between tree view and list view is done using checkbox (44). When a file is selected in (41) information on this file is displayed in the information text area (42). Information on the full filename is available. The full path name can be toggled using checkbox (43). A selected file can be visualized in the main user interface by pressing the display-button (45).

Settings dialog

The application contains many settings not suitable for display in the main user interface. These settings can be viewed and manipulated in the settings dialog. The settings dialog is displayed in figure A.3 (right).

All available settings are displayed in a settings tree view (48). A short description of the selected setting is displayed in the description area (46) and the value of the setting can be manipulated with setting specific controls in the value area (47). A complete list of available settings is given in table A.3.

Application	
Dimension	
Width	Width of application window at start-up
Height	Height of application window at start-up
General	
MouseEnterFocus	Whether mouse enter events at visualization window cause the visualization window to get focus
IO	
OpenParseError	Whether incomplete visualization files from failed parse processes are opened
SaveSourceCode	Whether to save source code when saving visualization
Keys	
ScrollLocal	Whether scroll keys are directed to column under mouse cursor or to all columns

continued on next page

continued from previous page	
StatusBar	
DisplayFullPath	Whether to display full path of filename in the status bar
Visualization	
ColorProfile	
Desaturation	The percentage of desaturation when selecting "bring to fromt" in the color profile
Column	
MinWidth	The minimum width of a column
Border	
Width	Separating border-width of columns in multi-column view
Header	
Antialias	Whether to use anti-aliasing when rendering column headers
Colour	The colour of the column header
Height	The height of the column header in pixels
Font	
Colour	The colour of the header font
Height	The height of the header font
Contour	
Enabled	Renders construct contours
Cushion	
Enabled	Renders construct cushions
MarginRightExp	Right margin extension
Spotlight	
Colour	The colour of the spotlight
Text colour	The colour of the text displayed in the spotlight area

Table A.3: Settings

A.3.3 Queries

The information being visualized is syntactic information (types of constructs and construct locations). However, the fact base contains many more details. In particular semantic information such as type information. It is possible to query the fact base.

There are two query systems. The first query system colours the cushions being visualized according to a query. This query can be selected from a list of available queries by using control (13).

The second query system provided is based on construct selection. Constructs can be selected by left-clicking on a construct. Files can be selected by clicking on the column header. Multiple selections are also possible by holding the SHIFT key while clicking a construct.

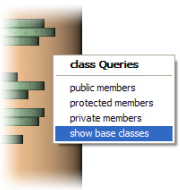


Figure A.4: Query select popup menu

Available queries for a certain selection are displayed when right-clicking on the current selection. This opens a popup menu which displays all valid queries. The query is executed by selecting

a query from the popup menu. An example popup menu for a class selection is displayed in figure A.4.

The result of a query is another selection and a text-string. The selection is displayed in the visualization window (25) and the text string is displayed in the output text area (23). When constructs in the query result are located in a file that is not being visualized the file will be visualized automatically such that the full selection is visible in the current visualization.

The selection from a query result may again be used in another query which allows for constructing cascaded queries. This is useful when navigating source code. An example is by running the "show base classes" query twice in order to navigate to the base classes of the base classes of the current class.

Appendix B

GCC C++ front-end modifications

Various modifications have been made to the GCC C++ compiler front-end. Table B.1 lists a short overview of all the modifications made to the C++ front-end. The modifications are logically grouped per added feature.

filename	**location in file**	**description**
Command-line switches		
c.opt	-	added C specific switches
common.opt	-	added common switches
opts.c	`common_handle_option`	added common switches
c-opts.c	`c_common_handle_option`	added C specific switches
c-opts.c	`c_common_missing_argument`	added switches that take arguments
gcc.h	`DEFAULT_WORD_SWITCH_TAKES_ARG`	added switches that take arguments
Compiler flags		
c-common.c	-	added compiler flags `xml_dump_filename` and `flag_xml_dump_local`
c-common.h	-	added compiler flags `xml_dump_filename` and `flag_xml_dump_local`
Include files/chains		
cpplib.c	`do_line`	removed call to _cpp_do_file_change
cpplib.c	`do_linemarker`	removed call to _cpp_do_file_change
cppfiles.c	`_cpp_find_file`	added search through `iwrapper` chain
cppfiles.c	`append_file_to_dir`	explicitly calculates `length` field
c-incpath.h	`head`, `tail`	added iwrapper chain head, tail and name
c-incpath.c	-	various changes for include wrapper
Preprocessor/Parser interface		
cpplib.h	`cpp_token`	added `line_end, col_end` fields
cpplex.c	`_cpp_lex_direct`	sets `line_end, col_end` fields
c-pragma.h	-	modified `c_lex_with_flags` declaration
c-lex.c	`c_lex_with_flags`	added line{start,end}, column{start,end} return values
c-lex.c	-	`get_line_from_map` function added
c-lex.c	`c_lex`	modified call to `c_lex_with_flags`
cp/parser.c	`cp_lexer_get_preprocessor_token`	handles line{start,end}, column{start,end} values

continued on next page

continued from previous page		
filename	**location in file**	**description**
cp/parser.c	`cp_token`	added new fields `line{start,end}`, `column{start,end}`
Syntax tree		
cp/parser.c	`c_parse_file`	added call to `xml_write`
cp/parser.c	-	`xml_save_prev_location` function added
cp/parser.c	-	`xml_save_location` function added
cp/parser.c	-	`xml_save_end_location` function added
cp/parser.c	various `cp_parser_*`	stores syntactic constructs
Preprocessor specific constructs		
cpplex.c	`_cpp_skip_block_comment`	stores C++ style block comment
cpplex.c	`skip_line_comment`	stores line comment
cppmacro.c	`cpp_get_token`	Store macro invocation calls
cpplib.c	`start_directive`	Store location of hash symbol
cpplib.c	`parse_include`	Store location of include directives

Table B.1: All changes made to the GNU compiler collection. All file locations are relative to the base directory `gcc-3.4.2/gcc`.

Various new functions and data types used by the fact extractor are located in additional files that have been added to the compiler front-end code base. The makefiles `Makefile.in` and `cp/Make-lang.in` of the compiler front end have been extended such that these additional files are compiled into the modified C++ compiler.

Appendix C

Fact extractor file format

The fact extractor fact base is represented in an XML formatted file. The grammar of the file format used in storing the fact base is given in section C.1. A simple example fact base is given in section C.2.

C.1 Grammar

The grammar of the XML representation of the fact base is given using a simple Extended Backus-Naur Form (EBNF) notation. Production labels are written in `CAPITALS` and string literals are written between single quotes.

A textual representation is a *well-formed* XML formatted representation of the fact base if, taken as a whole, it matches the production labelled `DOCUMENT`.

```
DOCUMENT  ::= HEADER ROOTOPEN ITREE STREE ROOTCLOSE
HEADER    ::= '<?xml version="1.0">'
ROOTOPEN  ::= '<factbase version="' STRING
              '" language="' STRING '" parser="' STRING '">'
ROOTCLOSE ::= '</factbase>'
```

The `STRING` production represents a valid XML formatted string. This means that it may, for example, not contain the < symbol.

The fact base contains references to a file for each syntactic construct. This file description is the file in which the construct is located and a stack of files that represent the order in which this file was included from other files (if so). The common way is to store these file descriptions in an attribute of the syntactic construct. However, this results in redundant data. Instead of storing the file description information at the syntactic construct a reference to this information is stored as an attribute of the construct. This reference points to a node in a tree which contains the inclusion order of *all* files present in the fact base. We call this tree the include tree `ITREE`. Each include file node `INODE` contains a line number which represents the line at which this file was included.

```
ITREE   ::= '<include_tree>' FNODE '</include_tree>'
FNODE   ::= '<file id="' NUMBER '" name="' STRING '">' INODE* '</file>'
INODE   ::= '<include_file id="' NUMBER '" name="' STRING '" line="' NUMBER '">'
            INODE* '</include_file>'
NUMBER  ::= ('1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9') DIGIT*
DIGIT   ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

The syntax tree, `STREE`, represents a tree that contains all syntactic constructs with additional semantic data stored as attributes. The tree is nested similar to the nesting structure of the constructs.

```
STREE     ::= '<syntax_tree>' SNODE '</syntax_tree>'
SNODE     ::= '<' TYPENAME ' ' ATTRIBUTE* '>' SNODE* '/>'
ATTRIBUTE ::= ('fid="' NUMBER '"') |
              ('loc="' NUMBER ':' NUMBER ':' NUMBER ':' NUMBER '"') |
              ('id="' NUMBER '"') |
              ('name="' STRING '"') |
              ('type="' TYPE '"') |
              ('kind="' KIND '"') |
              ('returns="' TYPE '"') |
              ('context="' TYPE '"') |
              ('f_extern="' BOOL '"') |
              ('f_const="' BOOL '"') |
              ('f_virtual="' BOOL '"') |
              ('f_purevirtual="' BOOL '"') |
              ('f_static="' BOOL '"') |
              ('f_artificial="' BOOL '"') |
              ('access="' ACCESS '"') |
              ('size="' NUMBER '"') |
              ('bases="' BASES '"')
BOOL      ::= '0' | '1'
BASE      ::= 'b(' NUMBER ',' TYPE ')'
BASES     ::= BASE BASE*
ACCESS    ::= 'public' | 'protected' | 'private'
KIND      ::= 'constructor' | 'destructor' | 'converter' |
              'operator_method' | 'operator_function' | 'method' |
              'function' | 'class' | 'struct' |
              'do' | 'for' | 'while' |
              'if' | 'switch'
TYPENAME  ::= 'simple_declaration' | 'compound_statement' |
              'init_declarator' | 'iteration_statement' |
              'member_declaration' | 'selection_statement' |
              'jump_statement' | 'expression_statement' |
              'labeled_statement' | 'try_block' | 'handler_seq' |
              'function_try_block' | 'exception_specification' |
              'member_specification' | 'function_body' |
              'using_directive' | 'namespace_body' |
              'using_declaration' | 'namespace_definition' |
              'enum_specifier' | 'explicit_specialization' |
              'base_clause' | 'class_head' | 'class_name' |
              'parameter_declaration_list' | 'parameter_declaration' |
              'parameter_declaration_clause' | 'statement' |
              'declaration' | 'block_declaration' |
              'condition' | 'identifier' | 'line_comment' |
              'block_comment' | 'function_definition_ad' |
              'include_directive_angled' | 'include_directive_quoted' |
              'macro_call' | 'unqualified_id' |
              'declarator' | 'primary_expression' | 'expression'
```

The type of a construct, `TYPE`, represents built-in types such as integers and floats but can also contain a reference to a user defined type. The reference is a number, `NUMBER`, which points to

a matching number of an 'id' attribute of a syntactic construct SNODE. Built-in types may have qualifiers QUAL. These qualifiers are restricted, volatile and constant represented by 'r', 'v' and 'c' respectively.

```
TYPE     ::= '::'
             'bool'
             'off(' TYPE ',' TYPE ')'
             'ptr(' TYPE ')'
             'ptr(' TYPE ',' TYPE ')'
             REAL
             REAL'(' QUAL ')'
             INT
             INT'(' QUAL ')'
             'void'
             'null'
             NUMBER

REAL     ::= 'float' | 'double' | 'ldouble'
INTID    ::= 'char' | 'short' | 'int' | 'long' | 'llong'
INT      ::= ('u' INTID) | INTID
QUAL     ::= 'rvc' | 'rv' | 'rc' | 'r' | 'vc' | 'v' | 'c'
```

To save memory space the TYPENAME string literals can be represented by their respective "short version" string literal. These literals are given next in the same order as the long version string literals:

```
TYPENAME  ::= 'sd' | 'cs' | 'ic' | 'is' | 'md' | 'ss' | 'js' |
              'es' | 'ls' | 'tb' | 'hs' | 'ft' | 'ex' | 'ms' |
              'fb' | 'ud' | 'nb' | 'ue' | 'nd' | 'en' | 'ec' |
              'bl' | 'ch' | 'cn' | 'pl' | 'pd' | 'pc' | 'st' |
              'dl' | 'bd' | 'co' | 'id' | 'lc' | 'bc' | 'fd' |
              'ia' | 'iq' | 'mc' | 'ui' | 'da' | 'pe' | 'ep'
```

C.2 Example

Following is a minimal example containing the following code fragment:

```
1  class A {
2    public:
3      A() {}
4  };
5
6  class B: public A {
7    public:
8      B() {}
9  };
10
11 int main(void) {
12   return 0;
13 }
```

The contents of the XML formatted file containing the fact base for this example code is listed below:

```
<?xml version="1.0" ?>
<factbase version="May 31 2005, 19:18:05" language="c/c++" parser="gcc/c++ v3.4.2">
<include_tree>
  <file id="0" name="example.cpp" />
</include_tree>
<syntax_tree>
<dl fid="0" loc="1:1:4:2" >
  <sd loc="1:1:4:2"  id="1">
    <ch loc="1:1:1:7"  id="2" name="A" kind="class">
      <id loc="1:7:1:7"  id="3" name="A"/>
    </ch>
    <ms loc="2:3:3:10" >
      <md loc="3:5:3:10"  id="4" kind="constructor" name="A" access="0" context="2">
        <cn loc="3:5:3:5"  id="5">
          <id loc="3:5:3:5"  type="5" id="6" name="A"/>
        </cn>
        <id loc="3:5:3:5"  type="5" id="7" name="A"/>
        <pc loc="3:7:3:7"  id="8"/>
      </md>
    </ms>
    <fd loc="3:9:3:10" >
      <fb loc="3:9:3:10" >
        <cs loc="3:9:3:10"  id="9"/>
      </fb>
    </fd>
  </sd>
</dl>
<dl fid="0" loc="6:1:9:2" >
  <sd loc="6:1:9:2"  id="10">
    <ch loc="6:1:6:17"  id="11" name="B" kind="class" bases="b(0, 2)">
      <id loc="6:7:6:7"  id="12" name="B"/>
      <bl loc="6:8:6:17"  id="13">
        <cn loc="6:17:6:17"  id="14">
          <id loc="6:17:6:17"  type="14" id="15" name="A"/>
        </cn>
      </bl>
    </ch>
    <ms loc="7:3:8:10" >
      <md loc="8:5:8:10"  id="16" kind="constructor" name="B" access="0" context="11">
        <cn loc="8:5:8:5"  id="17">
          <id loc="8:5:8:5"  type="17" id="18" name="B"/>
        </cn>
        <id loc="8:5:8:5"  type="17" id="19" name="B"/>
        <pc loc="8:7:8:7"  id="20"/>
      </md>
    </ms>
    <fd loc="8:9:8:10" >
      <fb loc="8:9:8:10" >
        <cs loc="8:9:8:10"  id="21"/>
      </fb>
    </fd>
  </sd>
</dl>
<dl fid="0" loc="11:1:13:1" >
  <sd loc="11:1:13:1" >
    <ic loc="11:5:13:1"  id="22" kind="function" name="main" returns="int" context="::">
      <id loc="11:5:11:8"  type="" id="23" name="main"/>
```

```
      <pc loc="11:10:11:13"  id="24"/>
      <fd loc="11:16:13:1" >
        <fb loc="11:16:13:1" >
          <cs loc="11:16:13:1"  id="25">
            <st loc="12:3:12:11" >
              <js loc="12:3:12:11"  id="26"/>
            </st>
          </cs>
        </fb>
      </fd>
    </ic>
  </sd>
</dl>
</syntax_tree>
</factbase>
```

Appendix D

Building

This appendix lists the method for compiling the fact extractor and the Code Structure Visualizer (CSV) from source code.

D.1 Fact extractor

D.1.1 Compiling

The fact extractor is based on the GCC C++ front-end version 3.4.2 [5]. To this end the source code for the front is needed:

- gcc-core-3.4.2.tar.bz2
- gcc-g++-3.4.2.tar.bz2

Some files of this source distribution are patched and some additional source files are needed. These files are located in the source distribution of CSV in the `gcc` directory.

To compile the fact extractor as a Microsoft Windows binary follow these steps:

1. Install Cygwin for windows from http://www.cygwin.com with the development packages (gcc compiler/make etc)
2. Extract the GCC C++ 3.4.2 front-end source code to a new directory; the files needed are

   - gcc-core-3.4.2.tar.bz2
   - gcc-g++-3.4.2.tar.bz2
3. Copy the patched files from the gcc directory in this directory to the gcc directory of the C++ front-end overwriting any existing files
4. Run cygwin; enter the gcc C++ front-end source code directory and run:

   ```
   ./configure && make
   ```
5. After the compilation has finished the file ./gcc/cc1plus.exe is the final fact extractor

D.1.2 Using the fact extractor outside the Cygwin environment

The fact extractor can now be used by the Code Structure Visualizer by copying the file ./gcc/cc1plus.exe to the CSV bin directory and renaming it to `cc1plus_struct.exe`.

In order to be able to run the Cygwin compiled fact extractor additional dll-files required to run Cygwin compiled executables need to be copied. The dll-files needed to run the executable depend on the version of Cygwin that was used; most likely the required dll-files are:

```
cygiconv-2.dll
cygintl-3.dll
cygwin1.dll
```

which are located in the Cygwin installation bin directory.

These dll-files must be placed in the same directory as the fact extractor executable.

D.1.3 Running the fact extractor

By placing the fact extractor in the CSV bin directory it is possible to run the fact extractor from within the CSV application. It is also possible to run the fact extractor from the command-line with the `gccfe.exe` executable. This executable serves as the front-end for the fact extractor.

D.2 Code Structure Visualizer

D.2.1 Compiling

The Code Structure Visualizer (CSV) depends on the following external distributions:

- libxml2 2.6.19 [10], this is an XML processor and it is used for processing the fact base
- iconv 1.9.1, the character encoding toolkit, part of libxml2
- zlib 1.2.2, the compression toolkit, part of libxml2
- wxWidgets 2.6.0 [14], cross-platform GUI toolkit

The windows version can be compiled using Microsoft Visual Studio 2003 or Microsoft Visual Studio 2005 (Beta). Project files for both compilers are available. In order to be able to compile this project the external distributions must be installed and directories to the include files of these distributions must be set in the project files. wxWidgets must be compiled as a dynamic link library. The project file generates two binaries. The `gccfe` binary and the CSV binary. The fact extractor binary needs to be copied in the CSV bin directory as this binary is not created by Microsoft Visual Studio (see section D.1.1).

Bibliography

[1] A.V. Aho, R. Sethi, and J.D. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley Publishing company, 1986. ISBN 0201100886

[2] Collard, M.L., Kagdi, H.H., and Maletic, J.I. (2003), "An XML-based Lightweight C++ Fact Extractor", Eleventh IEEE International Workshop on Program Comprehension, 10 - 11 May 2003, pp. 134 - 143, IEEE Computer Society Press, USA.

[3] CPPX, "CPPX - Open Source C++ Fact Extractor", see http://swag.uwaterloo.ca/~cppx/

[4] Cygwin, see http://www.cygwin.com/

[5] Free Software Foundation, GNU Compiler Collection (GCC), see http://gcc.gnu.org/

[6] Holt, R., A. Hassan, B. Laguë, S. Lapierre, and C. Leduc, "E/R Schema for the Datrix C/C++/Java Exchange Format", Proc. WCRE 00, IEEE CS Press, 2000, pp. 284-287.

[7] Holt, R. C., Winter, A., and Schürr, A., "GXL: Toward a Standard Exchange Format", in Proceedings of 7th Working Conference on Reverse Engineering (WCRE '00), Brisbane, Queensland, Australia, November, 23 - 25 2000, pp. 162-171.

[8] International C++ Standard, ISO/IEC 14882:1998(E)

[9] Kitware Inc., GCC-XML, see http://www.gccxml.org/

[10] libxml, "libxml - The XML C parser and toolkit of Gnome", see http://www.xmlsoft.org/

[11] OpenGL, Architecture Review Board, see http://www.opengl.org/

[12] The Visualization ToolKit, see http://www.vtk.org/

[13] van Wijk, J. J. and H. van de Wetering (1999). Cushion Treemaps: Visualization of Hierarchical Information. IEEE Symposium on Information Visualization (INFOVIS'99), San Francisco, CA, IEEE.

[14] wxWidgets, "wxWidgets - open source cross platform GUI toolkit", see http://www.wxwidgets.org/