Querying Large C and C++ Code Bases: The Open Approach

Alexandru Telea
Institute of Mathematics and Computer Science
University of Groningen, Netherlands
a.c.telea@rug.nl

Heorhiy Byelas
Institute of Mathematics and Computer Science
University of Groningen, Netherlands
h.v.byelas@rug.nl

Abstract

Static code analysis offers a number of tools for the assessment of complexity, maintainability, modularity and safety of industry-size source code bases. Most analysis scenarios include two main phases. First, the code is parsed and 'raw' information is extracted and saved, such as syntax trees, possibly annotated with semantic (type) information. In the second phase, the raw information is queried to check the presence or absence of specific code patterns which supports or invalidates specific claims on the code. Whereas parsing source code is largely standardized, and several solutions (parsers) exist already, querying the outputs of such parsers is still a complex task. The main problem resides in the difficulty of easily *translating high-level concerns in the problem domain into low-level queries into the raw data domain. We present here an open system for constructing and executing queries on industry-size C++ code bases. Our query system adds several so-called query primitives atop a flexible C++ parser, offers several options to combine these predicates into arbitrarily complex expressions, and has a very efficient way to evaluate such expressions on syntax trees of millions of nodes. We demonstrate the integration of our query system, C++ parser, and interactive visualizations, into the* SOLIDFX*integrated environment for industrial code analysis.*

1 Introduction

Static code analysis is one of the most powerful, scalable, robust, and accepted techniques for program understanding, software maintenance, reverse engineering, and reengineering activities. Static analysis encompasses a wide set of operations ranging from code parsing and fact extraction, fact aggregation and querying, up to interactive presentation. As compared to techniques based on formal methods, which attempt to prove the correctnes (or lack thereof) of the execution of a software system by checking it against a set of invariants, static code analysis has gained a wider acceptance, as the tools it involves have the scalability and maturity required to be applicable on industry-size code bases of millions of lines-of-code (LOC).

A typical static analysis pipeline would include three types of tools, as follows:

1. *Parsers* are used to analyze the original source code and produce a raw, low-level, representation thereof. This comes in most cases as a syntax tree, optionally annotated with type information.

2. *Query* engines are used to check the presence (or absence) of various facts in the code, by scanning the annotated syntax trees for the occurrence of corresponding patterns. Such queries can range from simple ones, *e.g.* "is a variable x of type T used in function f" up to sophisticated ones, *e.g.* "show all variables used before initialized" or "show the system's call graph". Queries are related to *metrics*, the latter being numerical values associated to specific code elements or patterns, such as cyclomatic complexity, code modularity, or class hierarchy depth.

3. *Presentation* engines are used to visualize the query results in context. These can be as simple as listing the query results, up to complex interactive visualizations of combinations of software graphs, source code, and code metrics.

In this paper, we focus on code analysis systems that combine the functionality of all three types of tools mentioned above for the C and C++ languages. C++ is one of the most widely spread programming language in the software industry. However, its complexity poses several problems to the construction of a truly powerful static code analyzer. While several C++ parsers exist, few are able to output complete annotated syntax trees, which are needed for a flexible query system. Secondly, query engines and parsers are often monolithically combined in a single tool. This is undesirable, as users require ways to design custom queries for custom problems. What one needs, is an *open query system* which offers

- a flexible, but simple to learn, way to construct a wide range of queries by assembling existing queries via some composition mechanism;

- an efficient execution for arbitrarily complex queries on real-world code bases whose syntax trees may have millions of nodes

Finally, as queries (and thus their results too) can be quite complex, a way is needed to let users both pose a query and examine its results in a simple and intuitive fashion.

In this paper, we present our experience in architecting such an open query system. We start with an existing C and C++ parser that generates full annotated syntax trees. Next, we design an open query system atop the parser's output, which satisfies our previously outlined requirements. Atop of these, we also add a query management mechanism consisting of query (de)serialization and query archiving in libraries. Finally, we integrate our query system in SOLIDFX, a fully fledged Interactive Reverse-engineering Environment (IRE) for C and C++, which combines parsing, querying, and data visualization and offers to reverse engineers the same look-and-feel that Integrated Development Environments (IDEs) such as Visual C++ or Eclipse offer to software developers.

This paper is structured as follows. In Section 2, we present related work in the context of interactive static analysis and reverse engineering, with a focus on C++. Section 3 describes the architecture of SOLIDFX in rough lines. We next detail the main components: the C++ parser, the query and software metric engine, and the data views, with a focus on the query system. Section 4 presents several applications of our tool on three real-life code bases. Section 5 discusses our experience with using SOLIDFX in practice and feedback obtained from actual customers. Section 6 concludes the paper with future work directions.

2 Previous Work

To understand the challenges of querying code during static analysis, we present a brief overview of results related to fact extraction, the fact querying proper, and fact visualization. In the following, we focus specifically on C++ static code analysis, since this is our target domain.

C++ static analyzers can be roughly grouped into two classes: *lightweight* analyzers do only partial parsing and type-checking of the input code, and thus produce only a fraction of the entire static information. Lightweight analyzers include SRCML [Collard et al. 2003], SNIFF+, GCCXML, MCC [Mihancea et al. 2007], and several custom analyzers constructed using the ANTLR parser-generator [Parr and Quong 1995]. Typically, such analyzers use a limited C++ grammar and do not perform preprocessing and scoping and type resolution. This makes them quite fast and relatively simple to implement and maintain. However, such analyzers cannot deliver the detailed information that we need for our queries, as we shall see later. Moreover, lightweight analyzers cannot guarantee the correctness of all the produced facts, as they do not perform full parsing. In contrast to these, *heavyweight* analyzers perform (nearly) all the steps of a typical compiler, such as preprocessing, full parsing and type checking, and hence are able to deliver highly accurate and complete static information. Well-known heavyweight analyzers with C++ support include DMS [Baxter et al. 2004], COLUMBUS [Ferenc et al. 2004], ASF+SDF [van den Brand et al. 1997], ELSA [McPeak], and CPPX [Lin et al. 2003]. However more powerful, heavyweight analyzers are also significantly (typically over one order of magnitude) slower, and considerably more complex to implement.

Heavyweight analyzers can be further classified into *strict* ones, typically based on a compiler parser which will halt on lexical or syntax errors (*e.g.* CPPX); and *tolerant* ones, typically based on fuzzy parsing or Generalized Left-Reduce (GLR) grammars, (*e.g.* COLUMBUS). Our early work to design an IRE for C++ used a strict `gcc`-based analyzer [Lommerse et al. 2005]. We quickly noticed the limitations of using strict analyzers. Typical users do not have a fully compilable code base, *e.g.* because of missing includes or unsupported dialects, but still want to be able to analyze it, or at least that subset which is parseable.

The output of static analyzers, mostly produced by running batches, can be fed to a range of *visualization* tools. Many such tools exist, ranging from line-level, detail visualizations such as SeeSoft [Eick et al. 1992] up to architecture visualizations which combine structure and attribute presentation, *e.g.* Rigi [Storey et al. 2000], CodeCrawler [Lanza 2004], or SoftVision [Telea 2004]. An extensive overview of software visualization techniques is provided by Diehl in [Diehl 2007].

Related to static queries, there is little available in terms of a generic, open query system for C++. Various analyzers, such as COLUMBUS and CPPX, provide a limited set of built-in queries, which aim to cover several code standards conformance and 'good coding practice' checks, *e.g.* that a baseclass should declare a virtual destructor, or that overriding a method should not change its access specifier. ASF+SDF goes probably the furthest here, proposing a formalism to define (and check) assertions on syntax trees. However, ASF+SDF is still far from full C++ support.

3 System Architecture

To understand the operation of the proposed open query system, described further in Sec. 3.3, we first outline the architecture of SOLIDFX, the Integrated Reverse-engineering Environment (IRE) into which the query system is combined with parsing and visualization. SOLIDFXis a commercial tool, the result of a design process of several years, combining our previous experience with a similar IRE called the Visual Code Navigator (VCN) [Lommerse et al. 2005] in several projects involving commercial, open-source, and academic C++ code, as well as our experience with using COLUMBUS.

In the design of SOLIDFX, we chose to use a *tolerant* extractor, as we noticed that users would not accept a tool that halts upon (trivial) syntax errors. Also, we chose a *heavyweight* extractor, for several reasons. First and foremost, we need all the facts in the code, *i.e.* a complete annotated syntax tree, in order to design an open query system, since we do not know upfront which facts one will need to include in one's queries. Also, in order to pose queries on-the-fly on source code and also present their results in code-level visualizations, we need fine-grained information such as the location, scoping, and types of each identifier in the source code. This level of detail requires a heavyweight extractor. Finally, the parsing, querying and presentation (visualization) are tightly connected in an easy-to-use environment via interactive, point-and-click, operations. This is required for an easy learning curve and favors users' acceptance.

This tight integration of parsing, querying, and visualization requires a fine-grained, but efficient, interface (API) to access the syntax tree, or *fact database*. Unfortunately, no heavyweight tolerant C++ extractor (coming in open-source form) that we were aware of offered such an interface, so we had to construct our own one. In the following we describe the extractor, query system, and data view components of our system, which are connected as described by the architecture shown in Figure 1.

3.1 Fact Database

The fact database contains all the static code data we work on. This includes the *raw facts*, produced by the extractor from the source code, as well as several derived facts, produced by the query and metric engines during the query process (Sec. 3.3). The database acts as a central repository, read and written by the various system components (Fig. 1).

To analyze a given code base, and produce the corresponding fact database, one must set up a so-called *project*. A project is much like a makefile, *i.e.* contains a set of source files, include paths, `#defines`, and language dialect settings. Users can create such projects either by hand, or by an automatic translation of makefiles or Visual C++ project files, following a technique similar to the so-called 'compiler wrapping' described for the COLUMBUS extractor [Ferenc et al. 2004]. For each source file (translation unit) in the project, the extractor parses and saves four kinds of data elements in the database: lexical, syntax, type, and preprocessor. Each data element is assigned a unique id. The database is structured as a set of binary files, one per translation unit.

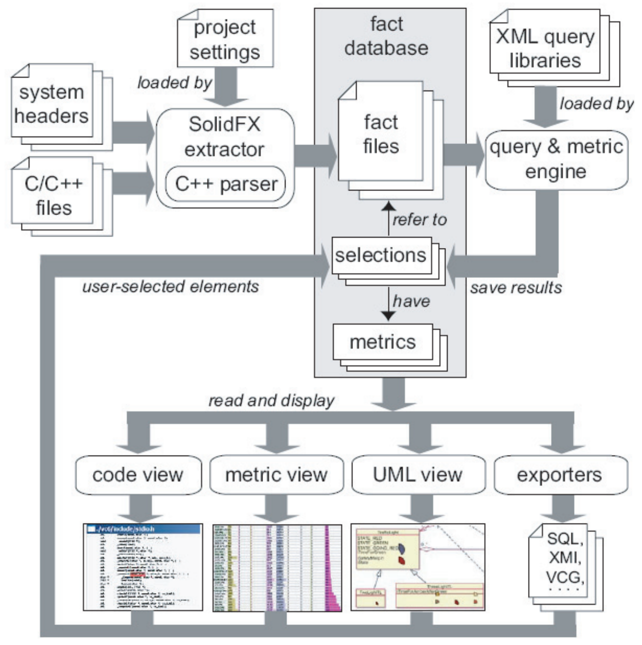


Figure 1. Dataflow architecture of SOLIDFX

The IRE components (parser, query engine, visualizations) communicate with each other by lightweight sets of ids, called *selections*, which resemble table views in a SQL database. The database creation, which involves parsing the source code, is by far the most consuming time of static analysis. After this process is completed, queries and visualizations only modify selections, a process which can be done at near-interactive rates (Sections 3.3,3.4)

3.2 Extracting Facts

As outlined in Section **??**, we use a C and C++ heavyweight analyzer of own construction. We based this analyzer on ELSA, an existing C++ parser designed using a GLR grammar [McPeak]. Atop the parser which produces a parse forest of all possible input alternatives, ELSA adds the complex type-checking, scoping, and symbol lookup rules that disambiguate the input to yield an Annotated Syntax Graph (ASG). The ASG contains two types of nodes: abstract syntax tree (AST) nodes, based on the GLR grammar; and type nodes, which encode the type-information created during disambiguation, and are attached to the corresponding AST nodes.

Although powerful, ELSA lacks some of the features we require in our interactive reverse-engineering context (Section **??**. First, ELSA requires preprocessed input, so no preprocessor facts are extracted. Also, token-level information, such as exact (row,column) locations, necessary for the code-level visualization (Section 3.4), are missing. Second, error recovery lacks, so incorrect code causes parse errors. Third, ELSA dumps the entire parse tree of the fully preprocessed input, which causes unnecessary overhead, as explained later.

We have extended ELSA to eliminate all these limitations [Boerboom and Janssen 2006], as follows. Our extended fact extractor works in five phases (see Figure 2).

First, the parser, preprocessor and preprocessor filter operate in parallel. The preprocessor reads the input files and outputs a token stream enriched with (row,column) location data. For this, we can use a standard C preprocessor, *e.g.* as provided by the Boost or *libcpp* libraries), patched to output token locations along with the tokens.

The parser reads the token stream from the preprocessor as it performs reductions and builds the AST. We extended the ELSA parser in order to handle incorrect and incomplete input, as follows. When a parse error is encountered, we switch the parser to a so-called *error recovery* rule, which matches all incoming tokens up to the corresponding closing brace (if the error occurs in a function body or class declaration scope) or semicolon (if the error occurred in a method, namespace, or global declaration scope). Besides skipping the erroneous code, we also remove the corresponding parts from the parse tree. The effect is as if the code block containing the error was not present in the input. This approach required adding only six extra grammar rules to ELSA's original C++ GLR grammar. Our approach, where error-handling grammar rules get activated on demand, resembles the hybrid parsing strategy suggested by [Knapen et al. 1999]. Compared to ANTLR, our method lies between ANTLR's basic error recovery (consuming tokens until a given one is met) and its more flexible parser exception-handling (consuming tokens until a state-based condition is met). All in all, our design balances well implementation simplicity with a good granularity of error recovery.

The error-recovery parsing is followed by ELSA's original AST dismbiguation and type-checking. Next, we filter the extracted facts (preprocessor, AST nodes, and type nodes) and keep only the facts which originate in, or are referred from, the project source files (Section 3.1). This eliminates all AST nodes which are present in the included *headers* but are not referred to by AST nodes contained in the analyzed *sources*, *i.e.* all declarations and preprocessor symbols contained in the include headers which are not referred to in the sources. Filtering the parsed output is essential for performance and scalability, as it reduces the output with one up to two orders of magnitude [1]. Finally, the filtered output is written to our database files using a custom binary format.

The several design choices made for the parser implementation, *i.e.* using the ELSA highly-optimized, hand-written, parser; providing lightweight error recovery at global declaration and function/class scope levels; filtering unreferenced symbols from the parser output; and writing the output in an optimized binary format, deliver a fast, tolerant, heavyweight parser. SOLIDFX is roughly three to six times faster than COLUMBUS, one of the fastest heavyweight C++ parsers that we could test, on projects of millions of lines of code [Boerboom and Janssen 2006]. For such projects, the fact databases saved on disk can take hundreds of megabytes. However, as we shall see next, querying such databases is still very fast.

3.3 Query and Metrics Engine

Now that we outlined the parser and fact database components, we can detail the query and metrics engine, the core of our static code analysis system.

Formally, a query implements the function

$$S_{out} = \{x \in S_{in} | q(x, p_i) = true\} \qquad (1)$$

[1]This is not surprising, considering that a typical "Hello world" program including `stdio.h` or `iostream` contains 100000 lines of code after preprocessing, of which only a tiny fraction is actually used

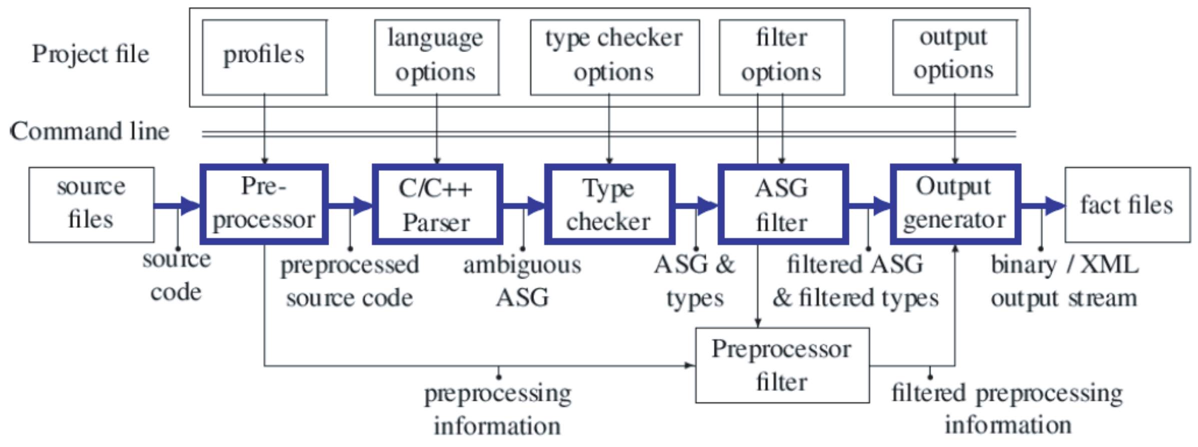


Figure 2. Architecture of the SolidFX C++ parser (main components shown in bold)

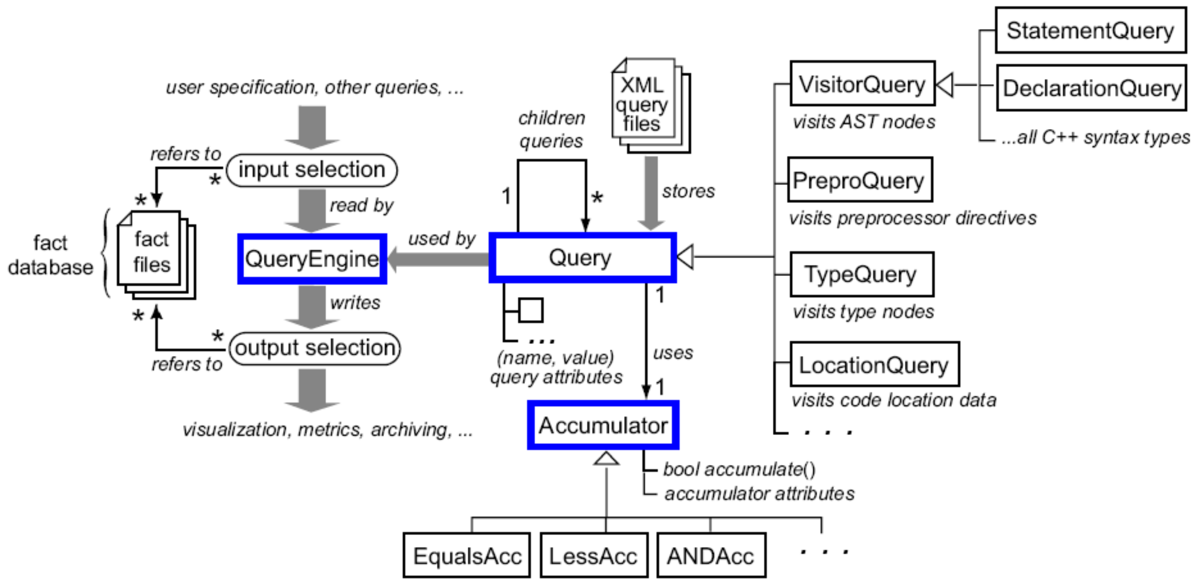


Figure 3. Query system architecture, with main elements marked in bold

that is, finds those elements x from a selection S_{in} which satisfy a predicate $q(x, p_i)$, where p_i are query-specific parameters.

The query engine is designed as a C++ class library which implements several specializations of the above query interface q, as follows (see also Fig. 3 which depicts the architecture of our query system).

For AST nodes x, VisitorQuery visits the tree rooted at x and searches for nodes of a specific syntax-type T, *e.g.* function, optionally checking for attributes, *e.g.* the functions name. For each of the approximately 170 syntax types T in our C++ GLR grammar, we generate a query class containing children queries for Ts non-terminal children and properties, or data attributes, for Ts terminal children. For instance, the Function query has a property $name$ for the functions name (which is an identifier, *i.e.* terminal, in the grammar), and two children queries $body$ and $signature$ for the functions body and signature (which are non-terminals). Besides queries for AST nodes, which search for syntax, we also created TypeQueries which search for type-data, and LocationQueries which search for code fragments having a particular file, line, or column location in the code.

Queries can be composed in query-trees. The query composition semantics is controlled by a separate customizable Accumulator class. When a child q_c of a query q yields a hit, q calls its Accumulators $accumulate()$ method, which returns true when the Accumulators condition has been met, else false. By default, all query nodes use an AND-accumulator, which returns true when all queries in a query-tree are satisfied. We designed Accumulator subclasses for different operators, *e.g.* AND, OR, $<$, $=$, and similar. These let us easily implement complex queries by combining simple ones. For example, to find all functions whose name begins with ”Foo” and have at least two parameters of type ”Bar”, we set the Function querys $name$ attribute to ”Foo*” (using regular expressions or wildcards), the $name$ attributes of the Type nodes of the functions $parameter$ children-queries to ”Bar”, and an AtLeastAccumulator with a default-value of 2 on the functions $signature$ child-query.

A given query tree q is applied on a given input element x by using the visitor pattern to find those elements y in the AST rooted at x matching the type of q's root, followed by an application of $q(y)$ based on recursion over q's children-queries. Overall, the query composition can be modified transparently by different accumulators, without having to change the query classes. We store query-trees in XML, and provide a query editor, so users can edit queries on-the-fly, without recompilation, and organize queries in custom query libraries. We have so far designed over 70 queries that cover a number of static analyses, such as identifying basic code smells *e.g.* case branches without break, class member initializations differing from the declaration order, changing access specification of a class member when overriden, base classes with constructed data members and no virtual destructors; and extracting class hierarchy, include, and call graphs. The query mechanism allows a flexible specification of a wide set of static queries, ranging from "find all variables called x" to "find all classes inheriting from $Base$ and containing a method which throws exactly two exceptions of type E". Several examples of queries and their applications are presented next in Section 4.

Queries can be executed on both in-memory and on-disk fact databases. On-disk queries are very efficient and have a negligible memory footprint, given the index maps that allow random access to elements-by-id and iterating over same-type elements (Sec. 3.1). We also implemented a cache mechanism which loads and keeps entire parsed translation units in memory on a most-recently-used policy. This improves query speed even further at the expense of more memory, roughly one megabyte per 5000 LOC. Another simple and effective speed-up uses early query termination when evaluating the query-tree accumulators. All in all, these mechanisms allow us to query millions of ASG nodes in a few seconds.

Several code metrics can be implemented directly using the query engine. For example, the metrics of the type "number of occurrences of code pattern P" can be implemented directly as

$$m(x) = |q(x, p_i)| \in \mathbb{R}, \forall x \in S_{in}. \qquad (2)$$

This associates a numeric value $m(x)$ to each element x of a selection S_{in} based on the number of hits of a corresponding query q which searches pattern P. Several metrics, such as McCabe's cyclomatic complexity, class interface sizes, coupling metrics, and most of the object-oriented metrics discussed in [Lanza and Marinescu 2006] can be implemented in this way.

3.4 Queries and the Data Views

The third and final component of SOLIDFX provides a set of interactive data *visualizations*, or views. These views serve both as input and output to the query operations: Users can select elements in the views and pass them as input to queries or metric engines, whose outputs can further serve as inputs for the views.

Figure 4 shows several data views. The *project view* lets users set up an analysis project, much like one sets up a build project in Visual Studio or Eclipse. The *output view* shows the fatabase files created by the parser, while the *selection view* shows all selections in the database. In this view, one can specify if the elements of a selection are to be shown in the other views, and if so, how to color them (as discussed next)[2]. The *query view* shows all available queries in the XML query library (Section 3.3).

To perform a query, *e.g.* "select all function definitions with n parameters", one selects the query in the query view, fills in the desired attributes, *e.g.* the value for n in the query GUI, and clicks on the selection to query in the selection view. A new selection, containing the query's output, is automatically added to the selection view. To browse the elements in a selection and their code metrics, if any, we provide a separate view called *selection monitor*, which uses the 'table lens' technique: a combination of text and colored bar graphs [Telea 2006]. When zoomed in, the table lens looks like a usual Excel table. When zoomed out, each row becomes a pixel row colored and scaled to show the data values, so the entire table becomes a set of vertical graphs.

Code views show the actual source code in the desired files. Selected code is highlighted in the respective selections' colors, thereby enabling one to spot the occurrence of particular events. We now see that, to construct such highlights, we need the exact (row,column) locations of every AST node and preprocessor directive from the parsing phase (Sec. 3.2). Code views can be zoomed out by decreasing the font size, thereby allowing one to overview larger amounts of code than using standard editors.

The *UML view* is a custom view showing UML-like class diagrams. The diagrams themselves are extracted from the fact database using queries which search for classes, inheritance relations, and associations. The latter can be defined *e.g.* as function calls, variable uses, or type uses. The extracted diagrams can be laid out by hand or automatically using the GraphViz library [AT & T 2007] or a custom graph layout library we developed. Moreover, class and method metrics can be drawn atop of the laid out diagrams using icons scaled and colored to show the metric values, following an extension of the technique described in [Termeer et al. 2005]. The combination of diagrams and metrics enables users to perform various types of code quality and modularity assessments (see Section 4).

Besides these built-in views, external visualization tools can be integrated within our IRE by writing appropriate data exporters. The inset in Fig. 4 shows such an external view which uses the SQL Lite database browser executable, with no modification, to visualize the data in a selection, *i.e.* the code element ids, their actual code, and the metrics computed on it, saved as a SQL database table by a data exporter. This type of integration allows us to extend our IRE by reusing several existing software analysis and visualization tools with a minimal amount of effort. External views are preferred when the interaction between the fact database and the view is rather loose, and when the amount of data to be passed to the view is limited, as compared to the built-in views, which heavily access the fact database at a fine-grained level.

4 Applications

We illustrate now the SOLIDFX IRE with several examples from a number of industrial projects[3]. In all these cases, the analyzed C++ code was developed by third parties, and we were not familiar with the code or its purpose before the analysis[4]. The results of the analyses were discussed together with the stakeholders, mainly using the data views. An typical analysis session would take a few hours from the initial code hand-over until the results were available. A complete code base assessment would typically take three to six such sessions, where increasingly refined questions and hypotheses would be tested during the later sessions by means of specific queries on narrowed-down parts of the code base.

[2]We recommend viewing this document in full color

[3]A video showing SOLIDFX in use is also available at `www.solidsource.nl/video/SolidFX/SolidFX.html`

[4]The only exception to this is the wxWidgets code base

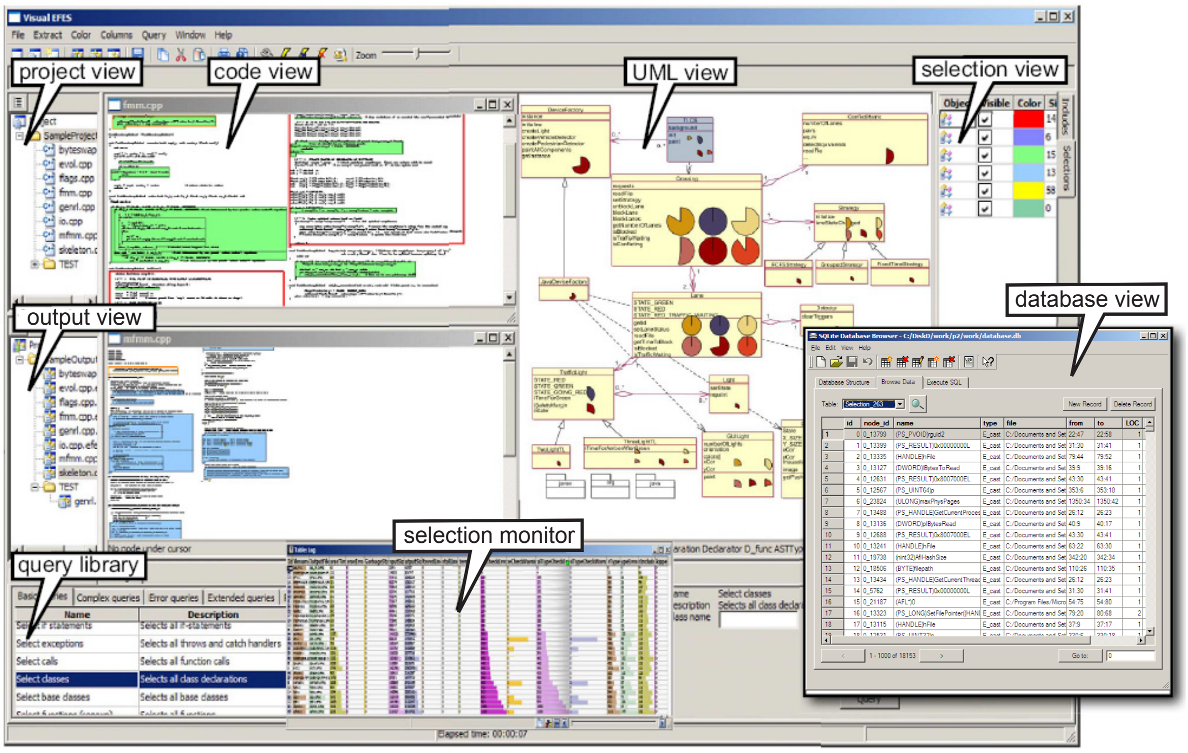


Figure 4. Overview of the SolidFX Integrated Reverse Environment

4.1 Finding Complexity Hot-Spots

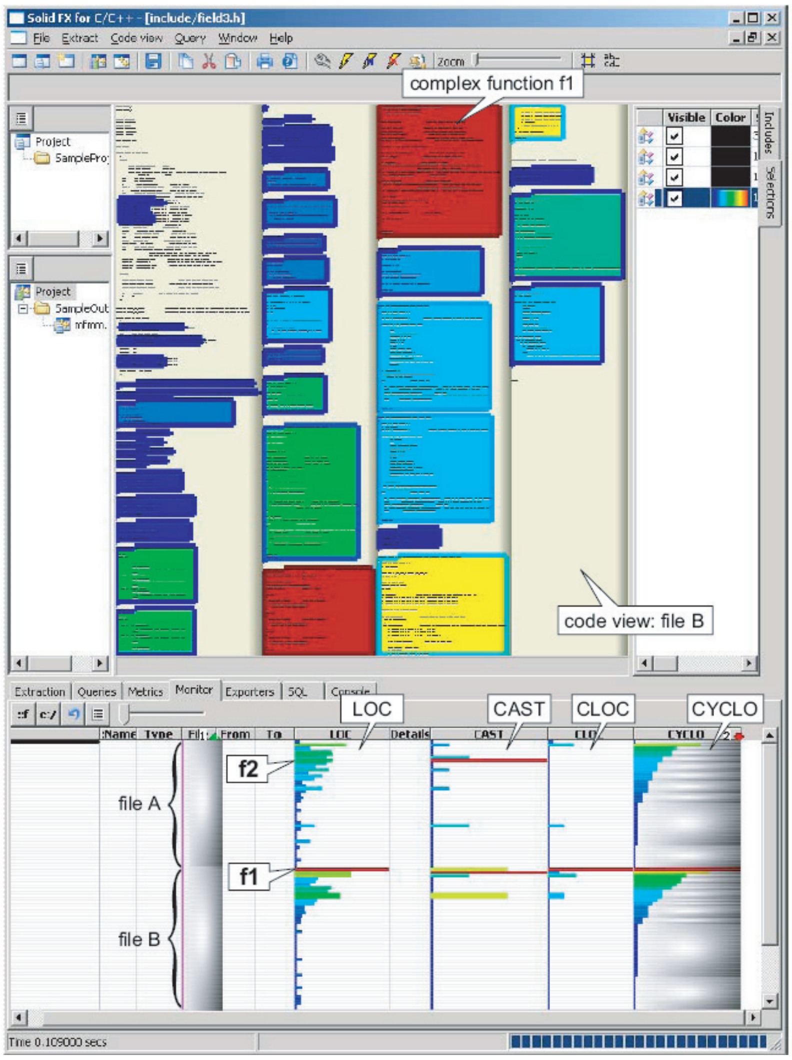
In the first application, we examine the complexity of the wxWidgets code base, a popular C++ GUI library having over 500 classes and 500 KLOC [Smart et al. 2005]. After extraction, we query all function definitions and compute several metrics on them: lines of code (LOC), comment lines ($CLOC$), McCabes cyclomatic complexity ($CYCLO$), and number of C-style cast expressions ($CAST$). Next, we group the functions by file and sort the groups on descending value of the $CYCLO$ metric, using the selection monitor widget. Figure 5 bottom shows a zoomed-out snapshot of this widget, focusing on two files A and B. Each pixel row shows the metrics of one function. The long red bar at the top of file B indicates the most complex function in the system (denoted $f1$). We also see that $f1$ is also the best documented (highest $CLOC$), largest (highest LOC), and, interestingly, in the top-two as number of C-casts ($CAST$). Clearly, $f1$ is a highly complex and important function in wxWidgets.

Double-clicking the table row of $f1$ opens up a code view showing all the selected function definitions and our clicked $f1$ flashing (Fig. 5 top, see also the video). The functions in the code view are colored to show two metrics simultaneously, using a blue-to-red colormap: the $CYCLO$ metric (highlight fill color) and the $CAST$ metric (highlight frame color). We see that $f1$ stands out as having both the body and frame in red, *i.e.* being both complex and having many casts. In the selection monitor, we also see that the function having the most casts, $f2$ (located in file A), is also highly complex (high $CYCLO$), but is barely commented (low $CLOC$). This may point to a redocumentation need.

4.2 Modularity Assessment

In this second application, the stakeholders were interested to assess the overall modularity of two given subsystems of a commercial database solution. The assessment was needed as a first step in a subsequent porting process. For this, we first extracted the static call graphs from the code, using a custom designed query that would look for function definitions and function calls, and link calls to the definitions using a technique which basically reproduces the working of a classical linker. Besides the call graphs, we also extract the system hierarchy, seen as methods-classes-files-folders. The call graph and hierarchy trees are next exported and visualized by Call-i-Grapher, a third-party tool designed to display large hierarchical graphs [Holten 2006]. The hierarchy is shown as a set of concentric rings, the sectors of which indicate methods, classes, and files (from inside to the outside) (Fig. 6). Call relations are drawn as splines, bundled to indicate relations emerging from, or going to, the same hierarchy ancestor.

The subsystem shown in Fig. 6 left is quite modular. We can easily discern the way its five subsystems call each other. Edge colors indicate the call direction: callers are red, callees are blue. We immediately see, for example, that *libraries* is only called from *database* and that emphcore does not call *libraries*. In contrast, the subsystem shown in Fig. 6 right, albeit of a similar size in terms of methods and classes, is far less modular. Here, we see two files which call each other in a highly complex way. There is little structure to see, so little hope that one can easily split these files into smaller loosely coupled units to simplify understanding and, later, porting. Here, we used the edge color to show the call type: green indicates static calls, whereas blue shows virtual calls. The blue edges appear to be somewhat bundled, so there is still some



Solid FX for C/C++ - [include/field3.h]
File Extract Code view Query Window Help
complex function f1
code view: file B
Visible Color
Includes
Selections
Extraction Queries Metrics Monitor Exporters SQL Console
LOC
CAST
CLOC
CYCLO
Name Type File From To LOC Details CAST CLO CYCLO
f2
f1
file A
file B
Time 0.109000 secs

Figure 5. Finding complexity hot-spots in the wxWidgets code base

hope we can locate some interface classes (containing mainly virtual functions) in this way.

4.3 Maintainability Assessment

In the third and last application, we are interested to assess the maintainability of a C++ code base implementing an UML editor using OpenGL, wxWidgets, and the STL library. The application was developed over a period of several years by three persons. The last developer, who worked for the second half of the period, did not have in the end a clear idea of the entire code architecture, and was concerned about the code maintainability. We started the analysis by extracting a number of class diagrams from the source code. The classes were loosely grouped into diagrams manually by the developer, based on his intuition and insight as to which belong together. As association relations, we considered method calls and referring to class types. Next, we computed three metrics on the methods: the lines-of-code (LOC), lines-of-comment-code ($CLOC$), and McCabe's cyclomatic complexity ($CYCLO$).

Figure 7 shows one of the extracted class diagrams, laid out automatically using GraphViz. Class heights are proportional to their methods' counts. Inheritance relations are drawn as black lines, while associations are drawn as light-gray lines (in order to reduce the visual clutter). On this picture, the architect recognized three main subsystems of the considered code base, along a Model-View-Controller pattern: the *data model*, containing the main application data structures; the *visualization core*, containing the control functions; and the *visualization plug-in*, containing rendering (viewing) functions. The diagram also shows that these subsystems are quite decoupled, which indicates a good maintainability. Further, we see the heavy use of several STL classes, mainly for the data model. This does not pose any maintenance problems, as it was decided to use STL in the system implementation from the beginning, and STL is stable and well-documented software.

Atop the class icons, the computed LOC and $CYCLO$ metrics are visualized using colored bar graphs. Long, red horizontal bars indicate high values. Thin, blue bars indicate low values. Within each class, the bar graphs are sorted in decreasing order of the $CYCLO$ metric. Looking at Fig. 7 top, we quickly discover

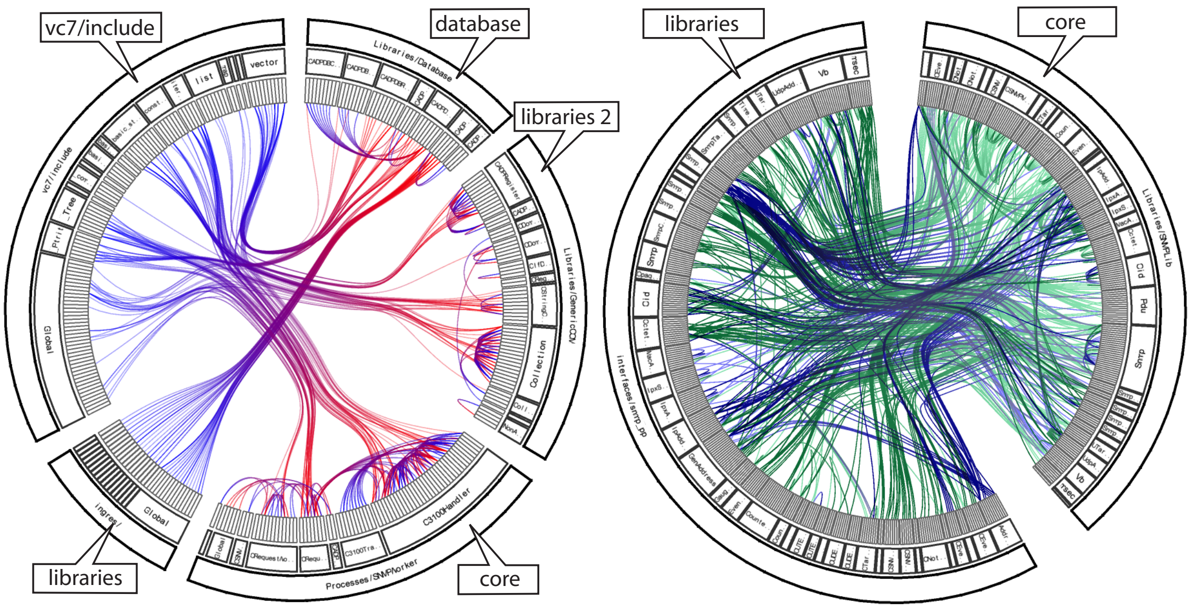


Figure 6. Call graph visualizations. Modular system (left) versus 'spaghetti code' (right)

an outlier class, marked X, in the visualization plug-in subsystem. This class has the highest $CYCLO$ and LOC values in the entire system, and has also many methods. All other classes have relatively small $CYCLO$ and LOC values, as indicated by the thin bars. Figure 7 bottom shows a zoomed-in view of the visualization plug-in. The sorted bar graphs, coupled with textual tooltips (not shown in the image), allow us to quickly locate the most complex methods, found of the class X, of the entire system. The most complex method has a McCabe value of 40, which is very large. Looking in detail at the code of X, we could later see that it was indeed very complex. However, the diagram shows us also that class X is *not* referred to directly from outside the visualization plug-in. Moreover, the lead developer recognized this class as containing his own code, which was indeed not yet cleaned up and refactored. Hence, although maintaining this class is indeed hard, this problem will not propagate to the entire system, but stays confined within the plug-in. Overall, it was assessed that the entire system is quite maintainable.

5 Discussion

At the current moment, SOLIDFX is a mature product capable of handling complex analyses and constructing visualizations of code bases of millions of LOC. Its design and evolution as a product started from a loosely coupled set of tools containing basic functionalities, such as preprocessing, parsing, fact database filtering, a query engine, and several software visualizations implemented using a range of techniques and languages. During numerous pilot projects, as early as [Telea 2004], we observed that some of the greatest obstacles in the acceptance of the proposed set of techniques were the high difficulty of setting up an analysis project, the steep learning curve of a set of hybrid tools, and the need to program (be it even only as scripting) in order to use a toolset.

The relative high success of SOLIDFX in the several recent projects we have used it is largely due to the high integration of its functions, presented under a uniform interface, and the possibility to execute complex analyses with a minimal, or no, amount of programming. Also, we noticed that using the IRE was not much more effective than using scripted command-line tools for the extraction phase, which is often performed in batch mode. However, for the exploration phase, the IRE and its tight tool integration were massively more productive than using the same tools standalone, connected by little scripts and data files.

However, besides the integration, the effectiveness of the presented solution relies heavily on the available query library. In most cases, users do not have the time (or expertise) to write custom queries from scratch, so they rely upon such queries to be readily available in the provided query libraries. To support this goal, we started designing our query libraries bottom-up, *i.e.* from simple, atomic, queries to more sophisticated, composite, ones. In this process, we noticed, however, that the purely context-independent, pattern-matching query engine that we initially designed, is not sufficient in all cases. For example, to find variables which are used before initialization, a more complex query mechanism involving *state*, as well as multiple traversals of the parse trees, is needed. Although such a mechanism can be implemented simply by coding the desired functionality in C++ atop of our parser, we are currently investigating more modular ways to integrate it within our open query API.

6 Conclusions

We have presented the design of an open query framework for static C++ code analysis and its integration in SOLIDFX, an Integrated Reverse Engineering environment for C and C++. The presented integration offers engineers a simple, but powerful, way to execute a number of code analyses pertaining to maintenance, refactoring, and software understanding, in a visual manner, by simple point-and-click operations on the code artifacts. Due to the high integration of querying with parsing and visualization, SOLIDFX enables users to conduct reverse-engineering sessions

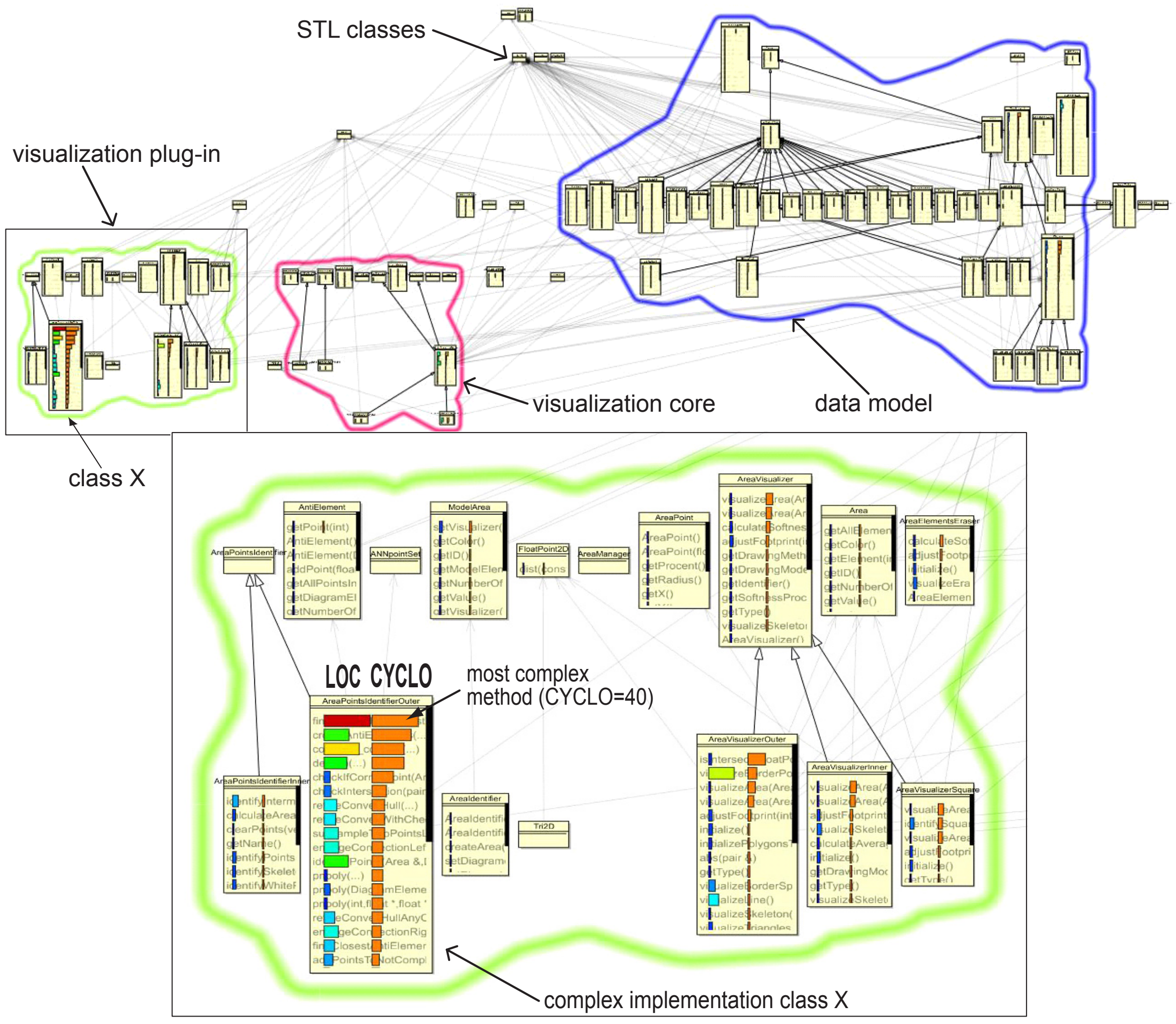


Figure 7. Maintainability assessment. Model-View-Controller architecture view (top). Zoomed-in view on the subsystem containing the most complex class (bottom)

with the same ease as software development is traditionally done in IDEs. Several typical applications of the IRE are presented.

We are currently working in extending SOLIDFX in several directions. Refined static information can be queried from the basic facts, such as control flow and data dependency graphs, leading to more complex and useful safety analyses. Secondly, we are working to implement a number of predefined ready-to-use analysis packages atop of our query system, such as checking for the MISRA C Standard [MISRA Association 2008], thereby making the use of SOLIDFX even more productive and easy.

References

AT & T. 2007. GraphViz. `www.graphviz.org`.

BAXTER, I., PIDGEON, C., AND MEHLICH, M. 2004. DMS: Program transformations for practical scalable software evolution. In *Proc. ICSE*, 625–634.

BOERBOOM, F., AND JANSSEN, A. 2006. Fact extraction, querying and visualization of large c++ code bases. MSc thesis, Eindhoven Univ. of Technology.

COLLARD, M. L., KAGDI, H. H., AND MALETIC, J. I. 2003. An XML-based lightweight C++ fact extractor. In *Proc. IWPC*, IEEE Press, 134–143.

DIEHL, S. 2007. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer.

EICK, S., STEFFEN, J., AND SUMNER, E. 1992. Seesoft - a tool for visualizing line oriented software statistics. *IEEE Trans. Soft. Eng. 18*, 11, 957–968.

FERENC, R., SIKET, I., , AND GYIMÓTHY, T. 2004. Extracting facts from open source software. In *Proc. ICSM*.

HOLTEN, D. 2006. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. In *Proc. InfoVis*, 741–748.

KNAPEN, G., LAGUË, B., DAGENAIS, M., AND MERLO, E. 1999. Parsing C++ despite missing declarations. In *Proc. IWPC*, 114–122.

LANZA, M., AND MARINESCU, R. 2006. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer.

LANZA, M. 2004. CodeCrawler - polymetric views in action. In *Proc. ASE*, 394–395.

LIN, Y., HOLT, R. C., AND MALTON, A. J. 2003. Completeness of a fact extractor. In *Proc. WCRE*, 196–204.

LOMMERSE, G., NOSSIN, F., VOINEA, L., AND TELEA, A. 2005. The visual code navigator: An interactive toolset for source code investigation. In *Proc. InfoVis*, 24–31.

MCPEAK, S. Elkhound: A fast, practical glr parser generator. Computer Science Division, Univ. of California, Berkeley. Tech. report UCB/CSD-2-1214, Dec. 2002.

MIHANCEA, P., GANEA, G., VEREBI, I., MARINESCU, C., AND MARINESCU, R. 2007. McC and Mc#: Unified C++ and C# design facts extractors tools. In *Proc. SYNASC*, 101104.

MISRA ASSOCIATION. 2008. Guidelines for the use of the C language in critical systems. `www.misra-c2.com`.

PARR, T., AND QUONG, R. 1995. ANTLR: A predicated-LL(k) parser generator. *Software - Practice and Experience 25*, 7, 789–810.

SMART, J., HOCK, K., AND CSOMOR, S. 2005. *Cross-Platform GUI Programming with wxWidgets*. Prentice Hall.

STOREY, M. A., WONG, K., AND MÜLLER, H. A. 2000. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming 36*, 2, 183207.

TELEA, A. 2004. An open architecture for visual reverse engineering. In *Managing Corporate Information Systems Evolution and Maintenance (ch. 9)*, Idea Group Inc., 211–227.

TELEA, A. 2006. Combining extended table lens and treemap techniques for visualizing tabular data. In *Proc. EuroVis*, 5158.

TERMEER, M., LANGE, C., TELEA, A., AND CHAUDRON, M. 2005. Visual exploration of combined architectural and metric information. In *Proc. VISSOFT*, 21–26.

VAN DEN BRAND, M., KLINT, P., AND VERHOEF, C. 1997. Reengineering needs generic programming language technology. *ACM SIGPLAN Notices 32*, 2, 54–61.