Architecting an Open System for Querying Large C and C++ Code Bases

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

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

Lucian Voinea
SolidSource BV
Eindhoven, the Netherlands
lucian.voinea@solidsource.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. Typically, such scenarios include three main phases. First, the code is parsed and 'raw' data is extracted and saved, such as syntax trees, possibly annotated with semantic (type) information. In the second phase, the raw data is queried to check the presence or absence of specific code patterns which supports or invalidates specific claims on the code. In the third and last phase, the query results are presented (visualized) such that correlations between code structure and query results are emphasized in an easily understandable way. Whereas parsing source code is largely standardized, using several existing parsers, querying the outputs of such parsers is still a complex task. The main problem resides in the difficulty of easily *translating high-level, cross-cutting concerns in the problem domain into queries in the raw data domain. We present here an open framework 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 options to combine these primitives into arbitrarily complex expressions, has a highly efficient way to evaluate such expressions on syntax trees of millions of nodes, and presents the query results in a visual, compact, intuitive way. We demonstrate our query framework, integratd in the* SOLIDFX *C++ reverse-engineering environment, with several real-world analyses on industrial codebases.*

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. In contrast to dynamic (run-time) analysis techniques, which require program compilation, instrumentation and execution and a suitable selection of input data, static analysis can be applied directly on, and needs solely, the source code of a system. Static analysis can support a wide range of code maintainability, quality, and safety assessments, based on methods such as dependency and impact analysis, type inference, and program slicing. Static code analysis is gaining wider acceptance, as the tools it involves are reaching 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 includes three types of tools, as follows:

1. *Parsers* are used to analyze the input source code and produce a raw, low-level, representation thereof. This comes usually 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.* "select all variables used before initialized" or "extract the system's call graph". Queries are related to *metrics*, *i.e.* numerical values associated to code elements, *e.g.* cyclomatic complexity,

code modularity, or class hierarchy depth[1].

3. *Presentation* engines are used to visualize the query results in context. These range from simple tabular listings of the query results, up to complex interactive visualizations of multiple attributes such as software graphs, source code, and metrics.

In this paper, we focus on users interested in static analysis for the C and C++ languages. C++ is one of the most widely spread programming language in the software industry [1]. However, its complexity poses several non-trivial problems to the construction of a truly *effective* static C++ analyzer. Considering the static analysis pipeline mentioned above, these problems are:

1. *Parsers:* While several C++ parsers exist, few can output complete annotated syntax trees. Complete trees are required for a flexible query system.
2. *Query:* Query engines and parsers are often monolithically merged in a single tool. However, users need custom queries for custom problems. We need an *open query system* with
   - a flexible, but simple to learn, way to build a wide range of queries by composing existing queries;
   - efficient execution for arbitrarily complex queries on code bases of millions of LOC;
3. *Presentation:* Queries and their results can be quite complex, so users need ways to pose a query on some given code and examine its results easily and intuitively. Ideally, we would like a *click-to-query* system working directly with the source code in an editor.

In this paper, we present the design challenges, architecture, implementation, and use of such an open query system. We start with an existing C++ parser that generates syntax trees annotated with type information. Secondly, we extend the parser to make it applicable in an interactive query context, and design an open query system atop the parser's output, which satisfies the requirements outlined above. Next, we 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 code analysis and visualization, offering 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 Sec. 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 and metric engines. Section 4 presents several applications of our query and metric engines on three real-life code bases. Section 5 discusses our experience with using our solution in industrial practice and feedback obtained from actual users. Section 7 concludes the paper with future work directions.

[1]Since we are using a common engine to compute queries and metrics (Sec. 3.3), following statements about queries also apply to metrics unless specifiedotherwise.

2 Previous Work

To understand the challenges of interactively querying C++ code during static analysis, we present a brief overview of results related to fact extraction, the fact querying proper, and fact visualization, with a focus on C and C++. We follow the three-stage pipeline as in Sec. 1.

2.1 Parsers

C++ parsers can be roughly grouped into two classes: *Lightweight* parsers do only partial parsing and type-checking of the input code, and thus produce only a fraction of the entire static information. These include SRCML [2], SNIFF+, GCCXML, MCC [3], and several custom analyzers constructed using the ANTLR parser-generator [4]. Typically, such analyzers use a limited C++ grammar and do not perform preprocessing, scoping, type resolution, and overloading. This makes them quite fast and relatively simple to implement and maintain. However, such analyzers simply cannot deliver the detailed information that we need for our queries, as we shall see later. Worse, lightweight analyzers cannot guarantee the correctness of all the produced facts, as they do not perform full parsing and/or type analysis.

In contrast to these, *heavyweight* parsers perform (nearly) all the steps of a typical compiler, except code generation, and hence are able to deliver highly accurate and complete static information. Well-known heavyweight analyzers with C++ support include DMS [5], COLUMBUS [6], ASF+SDF [7], ELSA [8], the EDG front-end [9], and CPPX [10]. However more powerful, heavyweight analyzers are also significantly (typically over one order of magnitude) slower, considerably more complex to implement, and hardly customizable.

Heavyweight analyzers can be further classified into *strict* ones, typically based on a compiler parser which halts on lexical or syntax errors (*e.g.* CPPX); and *tolerant* ones, typically based on fuzzy or Generalized Left-Reduce (GLR) parsing, which do a tolerant parsing followed later by the strict disambiguation and type checking (*e.g.* COLUMBUS). Our earlier work to design the Visual Code Navigator, an interactive query tool for C++, used a strict `gcc`-based parser [11]. We quickly noticed the practical limitations of strict parsers: Many users do not have a fully compilable code base, due to missing headers, unsupported dialects, or simply errors in the code. Yet, one still wants to be able to query such code, or at least its parseable subset. Consequently, we concluded that a tolerant C++ analyzer is to be preferred.

2.2 Query engines

There is very little available in terms of a generic, open static 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. Abxsoft's CodeCheck [12] offers a scripted C-like query language. Although flexible, the query language effectiveness is bounded by the quite limited set of facts that can be checked, which is in turn limited by the built-in parser. ASF+SDF goes probably the furthest in query design flexibility, proposing a formalism to define (and check) complex assertions on syntax trees. However, ASF+SDF is still very far from full C++ support - for example, it does not offer complete lookup and scoping. Also, its generic character makes its rapid applicability to the complex specifics of C++ quite challenging.

None of the above query systems is directly integrated with interactive data presentation. Queries are posed in batch-mode, using script files, which makes rapid 'what-if' exploration of large code bases difficult and time-consuming. A second serious problem of the current state-of-the-art in static analysis is the extremely limited amount of detailed implementation information that many papers provide. This is especially true for C++ analyzers, which are notoriously complex. However, such 'details' are of paramount importance when implementing a complete, efficient static query system, as we shall see in Sec. 3 and 3.3.

2.3 Presentation

The most frequent way to present static analysis query results is in tabular or (hyper)textual form [13, 6]. Visualization tools are more effective than plain text, as they can depict higher amounts of information, and also multivariate and/or relational information. Many visualization tools exist, ranging from line-level, detail visualizations such as SeeSoft [14] up to architecture visualizations which combine structure and attribute presentation, *e.g.* Rigi [15], CodeCrawler [16], or SoftVision [17]. An extensive overview of software visualization techniques is provided by Diehl in [18].

A recent attempt to combine visualization and C++ static analysis is the SOLIDFX reverse-engineering framework [19]. SOLIDFX has several advantages compared to other similar solutions:

- is based on a heavyweight, tolerant, and extensible C++ parser;
- offers both line-level and architecture-level visualization plug-ins;
- is easily extensible with new analysis plug-ins.

Given the above, we chose to integrate our open C++ query system in SOLIDFX. The integration process, and its results, are detailed next.

3 System Architecture

To understand the operation of our proposed open query system, described next 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 [19], the result of a design process of several years, combining our previous experience with a similar IRE called the Visual Code Navigator (VCN) [11] in projects involving commercial, open-source, and academic C++ code, as well as our experience with COLUMBUS.

SOLIDFXuses a *tolerant* parser, as we noticed that most users would not accept a tool that halts upon (trivial) syntax errors. A *heavyweight* parser was chosen, for several reasons. First and foremost, we need all the facts in the code, *i.e.* a complete syntax tree annotated with type information, 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. Secondly, in order to pose queries on-the-fly on source code and also present their results in code-level visualizations, *e.g.* click-to-query in a code editor, we need fine-grained information such as the location, scope, and type of each code identifier. This requires a heavyweight extractor.

A second consequence of this tight integration of parsing, querying, and visualization required by a click-to-query tool, we need a fine-grained *and* efficient interface (API) to access all the parsed static information on-the-fly. Unfortunately, no heavyweight tolerant C++ extractor that we were aware of offered such an interface, so we had to build one atop of our tolerant C++ parser. In the following, we describe the design of our query system and its tight integration with the C++ parser and data visualization in the SOLIDFXenvironment. We refer to Figure 1 which shows the arcitecture of our system and its four main elements: the fact database, the fact extractor, the query and metric engine, and the visualization components.

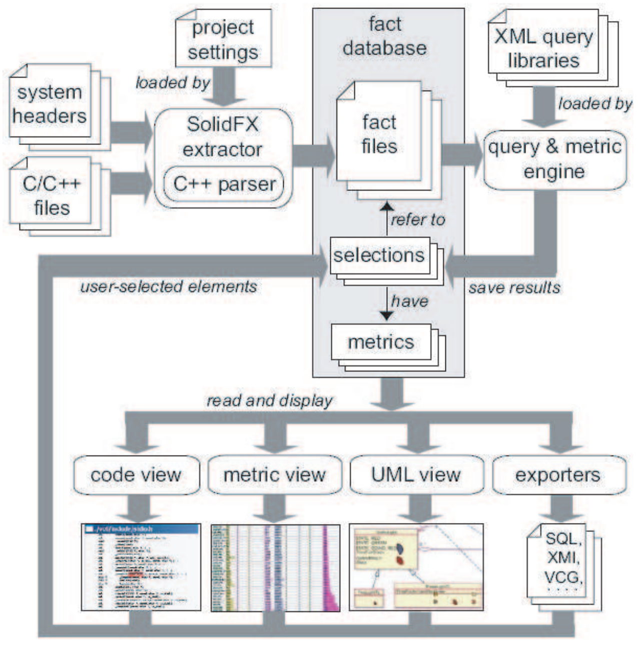

Figure 1. Dataflow architecture of SOLIDFX

3.1 Fact Database

All parsed or queried static data is stored in a so-called *fact database*. This includes the *raw facts*, produced by the parser from source code, but also *derived* facts, produced by the query and metric engines (Sec. 3.3).

A fact database is created by analyzing a given code *project*. Similar to a makefile, a project contains a set of C/C++ files, include paths, preprocessor defines, and language dialect settings, and is created either by hand or by automatic translation of makefiles or Visual C++ project files, using a technique similar to the 'compiler wrapping' described for COLUMBUS [6]. For each source file (translation unit), the parser saves four kinds of data elements in the database: syntax, type, preprocessor, and location. Each data element is assigned a unique id. The database is structured as a set of binary files, one per translation unit.

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 database creation, queries and visualizations do not change the annotated syntax trees, but only modify selections, a process which can be done at near-interactive rates (Sec. 3.3,3.4). To be concrete, queries ranging from simple ones ("select all functions whose name matches a regular expression") up to complex ones ("extract a call graph involving only non-virtual functions") take between one and 3-4 seconds on a code base of a few hundred thousands lines of code on a standard 4MB 2.2 GHz PC. The (small) speed fluctuations are mainly dependent on the database file caching performed by the SQL engine and the operating system. Moreover, it is illustrative to note that *complex* queries take *less* time, since they have stricter conditions which lead quicker to early query termination, and also generate less database traffic.

3.2 Extracting Facts with an Extended Parser

As outlined in Sec. 3, we use a C and C++ heavyweight analyzer of own construction. We based our analyzer on ELSA, an existing C++ parser designed using a GLR grammar [8]. We chose ELSA as it is the only open-source heavyweight tolerant C++ analyzer we are aware of. ELSA produces a parse forest of all possible input alternatives, which are next disambiguated to a single Annotated Syntax Graph (ASG) using the C++ scoping and lookup rules. In the disambiguation phase, type information is added to the parse tree, *i.e.* information linking each symbol with its declaration. The ASG contains two types of nodes: abstract syntax tree (AST) nodes, creating during parsing; and type nodes, created during disambiguation, which are attached to the typed AST nodes.

Although it comes closest to our architectural and user requirements outlined earlier in this section, ELSA still lacks features needed in our interactive click-to-query setup (Sec. 3). These limitations are as follows:

- *L1:* ELSA requires preprocessed input, so it cannot understand or query preprocessor facts;
- *L2:* Exact (row, column) locations for all AST nodes, needed for click-to-query, are lacking;
- *L3:* Error recovery is lacking, so incorrect code generates no output at all.
- *L4:* ELSA dumps the entire AST of its input, which causes large overhead making real-time querying impossible;

We have extended ELSA to eliminate limitations $L1 - L4$, as described next. Our extended C++ extractor works in five phases (see Figure 2 and [20] for full details).

First, the parser reads the token stream from the lexer as it performs reductions and builds the AST. Traditionally, the lexer of a C parser would simply get tokens from an already postprocessed file. In our setup, however, the lexer gets tokens by *directly* calling back the preprocessor, which preprocesses the input on-the-fly. For this, we succesfully used both the Boost [21] and *libcpp* preprocessors, which we patched to output token locations along with the tokens and also to save preprocessor information in the fact database. This setup barely modifies the original ELSA parser, and addresses limitation $L1$ and $L2$.

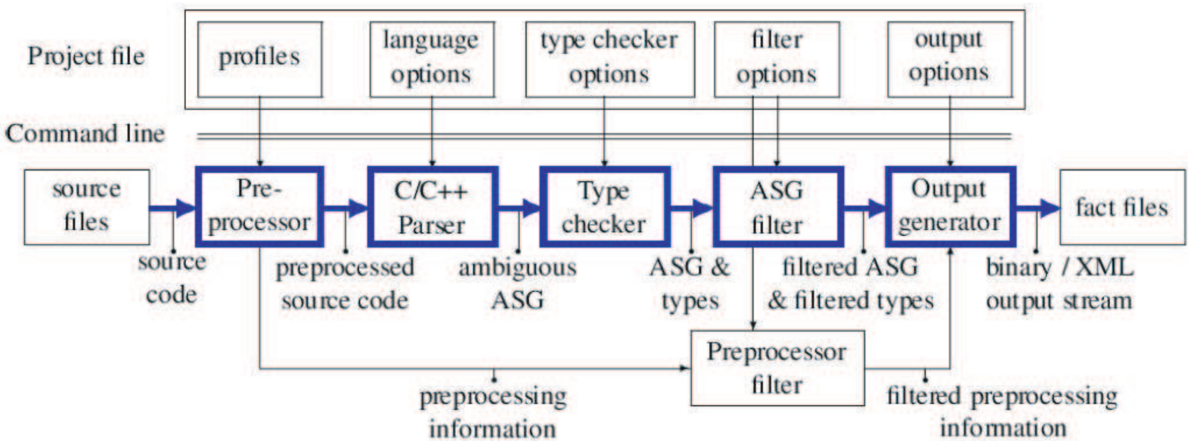

Figure 2. Architecture of the SOLIDFX C++ parser (main components shown in bold)

A second extension of ELSA allows us to handle incorrect and incomplete C++ input, as follows. When a parse error is encountered, we switch the parser to a so-called *error recovery* grammar rule, which will match 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 occurs in a method, namespace, or global-declaration scope). Besides skipping the erroneous code, we also remove the corresponding parts from the AST. The net effect is as if the code containing the error up to the matching '}' or ';' was not present in the input. This solution 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 [22]. 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). This design balances well implementation simplicity with a good error-recovery granularity, thereby addressing limitation $L3$, and adds less than 10% overhead to the parsing.

The error-recovery-enhanced parsing is followed by ELSA's original AST dismbiguation and type-checking. Next, we filter the extracted preprocessor, AST, and type nodes, and keep only those which originate in, or are referred from, the project source files (Sec. 3.1). This eliminates all code from included *headers*, *e.g.* declarations and preprocessor symbols, which is not referred by code in the analyzed *sources*. Filtering the parsed output is essential for performance and scalability, as it reduces the output with one up to two orders of magnitude, and makes the fact database queryiable in near-real-time, as we shall see next[2].

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

Finally, the filtered output is written to the fact database using a custom binary format. Filtering effectively addresses limitation $L4$.

The several design choices made for the parser front-end, *i.e.* using the ELSA highly-optimized, hand-written, parser; providing 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, make our modified ELSA parser 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 [20].

3.3 Query and Metrics Engine

3.3.1 Preliminaries

The query and metrics engine is the core of our static code analysis system, and is described in detail next.

Formally, a query implements the function

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

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

3.3.2 Design and Implementation

Our query engine is designed as a C++ API (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).

There are four main subclasses of the Query interface: PreproQuery, TypeQuery, LocationQuery, and VisitorQuery. PreproQuery offers a simple way to search for specific preprocessor constructs, *e.g.* comments including

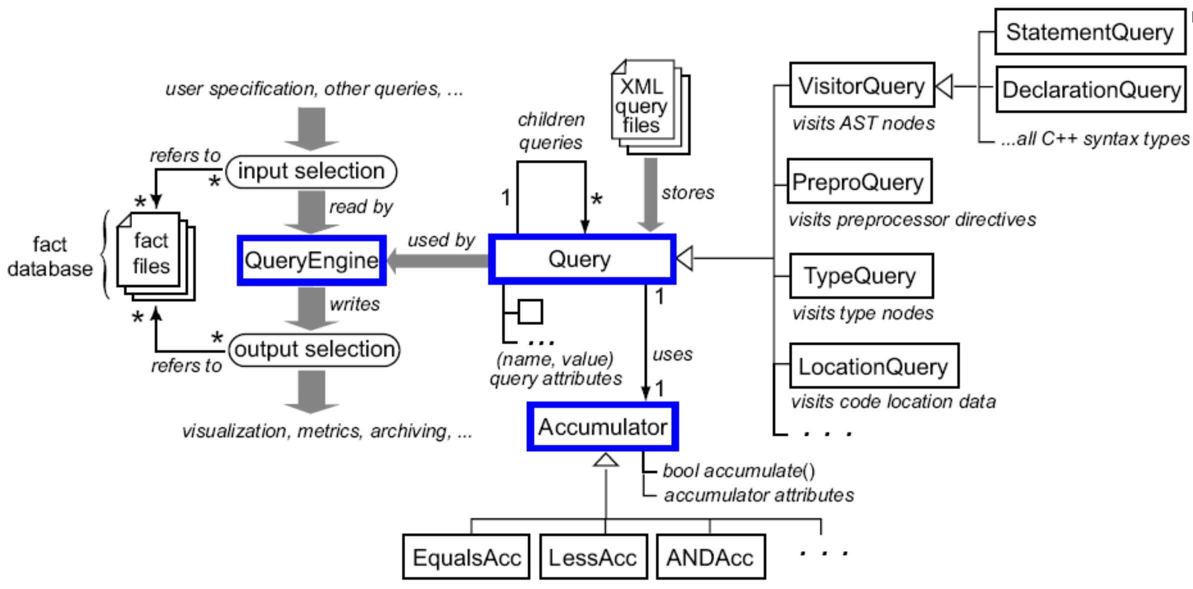

Figure 3. Query system architecture, with main elements marked in bold

a given text, macros, macro calls, or conditionals. Since the C preprocessor grammar is quite flat, PreproQuery has a straightforward implementation.

TypeQuery offers a way to search the type information produced during the disambiguation of an extracted ASG. This interface supports queries such as ”what is the type of this given variable?”, but also queries to determine the relation between two given types (identity, subtyping, subsuming, etc).

LocationQuery is a simple interface that queries all symbols within a given code location range (file, row, column).

The most complex (and useful) of the Query interface implementations is VisitorQuery. For AST nodes x, VisitorQuery visits the syntax tree rooted at x and searches for nodes of a specific syntax-type T, *e.g.* function, optionally checking for attributes, *e.g.* the function's name. For each of the approximately 170 syntax types T in our C++ GLR grammar [8], we generate a query-class containing children queries for T's non-terminal children and properties, or data attributes, for T's terminal children. For instance, the FunctionQuery has a property $name$ for the function's name (which is an identifier, *i.e.* terminal, in the grammar), and two children queries $body$ and $signature$ for the function's body and signature (which are non-terminals). All above query-classes are generated automatically from our C++ GLR grammar, using a modified version of the Elkhound parser-generator which comes along with Elsa [8]. In this way, any modification to the C++ grammar used is automatically reflected in the query API.

To perform more complex analyses, the above Query interfaces 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 Accumulator's $accumulate()$ method, which returns true when the Accumulator's condition has been met, else false. By default, all query nodes use an AND-accumulator, which returns true when all queries in its query-tree are satisfied. We implemented Accumulator subclasses for different operators, *e.g.* AND, OR, $<$, $=$, and similar. These let us easily implement complex queries by combining simpler ones. For example, to find all functions whose name begins with ”Foo” and have at least two parameters of type ”Bar”, we set the FunctionQuery's $name$ attribute to ”Foo*” (using regular expressions or wildcards), the $name$ attributes of the Type nodes of the function's $parameter$ children-queries to ”Bar”, and use an AtLeastAccumulator with a default-value of 2 on the function's $signature$ child-query.

As outlined earlier in this section, VisitorQuery applies a given query-tree q on a given input element x by using the visitor pattern to find those elements y in the AST rooted at x which match the type of q's root, followed by an application of $q(y)$ based on recursion over q's children-queries. This design decouples the search implementation from the specification of what to search for: The former is implemented in the VisitorQuery class, while the latter is implemented in the particular query-tree instance used. Overall, the query composition can be modified transparently by different accumulators, without having to change the query classes.

Although flexible, the VisitorQuery design outlined above cannot go beyond matching a semi-fixed structural pattern (the query-tree) on the input code (the AST). In some cases, one wants to query patterns which have a more variable structure, *e.g.* the fact that a variable has been initialized before being used. To support such queries, we added a set of *iterators* over the parsed AST to out query API. These offer inorder code traversal and full access to the additional type information. Using the iterators API, any static query that we are aware of can be implemented.

The system stores 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 API 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 in Sec. 4.

3.3.3 Performance considerations

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. However, in our click-to-query scenario (Sec. 3), we require near-real-time query response, even for large fact databases of hundreds of megabytes. To achieve this, we designed a few additional mechanisms, as follows.

Typical C++ syntax trees are shallow. A translation unit (the root of the syntax tree) contains thousands of relatively limited-depth subtrees, one per global symbol. The largest such trees occur for namespaces and class declarations, which in turn may contain hundreds of shallow subtrees for their symbols (*e.g.* methods in a class). Extensive testing confirmed that the largest part of the time spent in a query is in iterating over all these subtrees to find the requested one(s), during the VisitorQuery visit process described earlier. Hence, we can accelerate the query process by precomputing hash-maps that store all global symbols of a given kind, *e.g.* functions, class declarations, and global variables. We implemented this technique during the fact database extraction, right after the filtering and before serialization (Sec. 3.1). Using these precomputed hash-maps, VisitorQueries can now directly iterate over global-scope subtrees of a given kind, without having to visit the entire translation unit. Within the subtree of a global construct, the usual visiting process is used. This relatively simple optimization accelerates the query process by a factor between 8 and 10 on typical C++ code bases which include standard library headers, as these contain (tens of) thousands of global-scope symbols.

A second optimization provides 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 by a factor between 3 and 7, at the expense of more memory, roughly one megabyte per 5000 parsed-LOC. A third 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 on a 3GHz PC with 2 GB RAM. This gives us the desired performance for our interactive click-to-query scenarios.

3.3.4 Code Metrics

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 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. Interestingly, many typical static-analysis metrics, such as McCabe's cyclomatic complexity, class interface sizes, coupling metrics, and most of the object-oriented metrics discussed in [23] can be implemented in this way. For more complex metrics, one can always implement them by directly calling the query API described above.

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 click-to-select code fragments in the views and pass them as input to queries or metric engines, whose outputs can be further displayed in 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 fact database files created by the parser, while the *selection view* shows all selections in the database. In the selection view, one can specify which selections are to be shown in the other views and how to color their elements (as discussed next)[3]. The *query view* shows all available queries in the XML query library (Sec. 3.3).

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

[3]We recommend viewing this document in full color

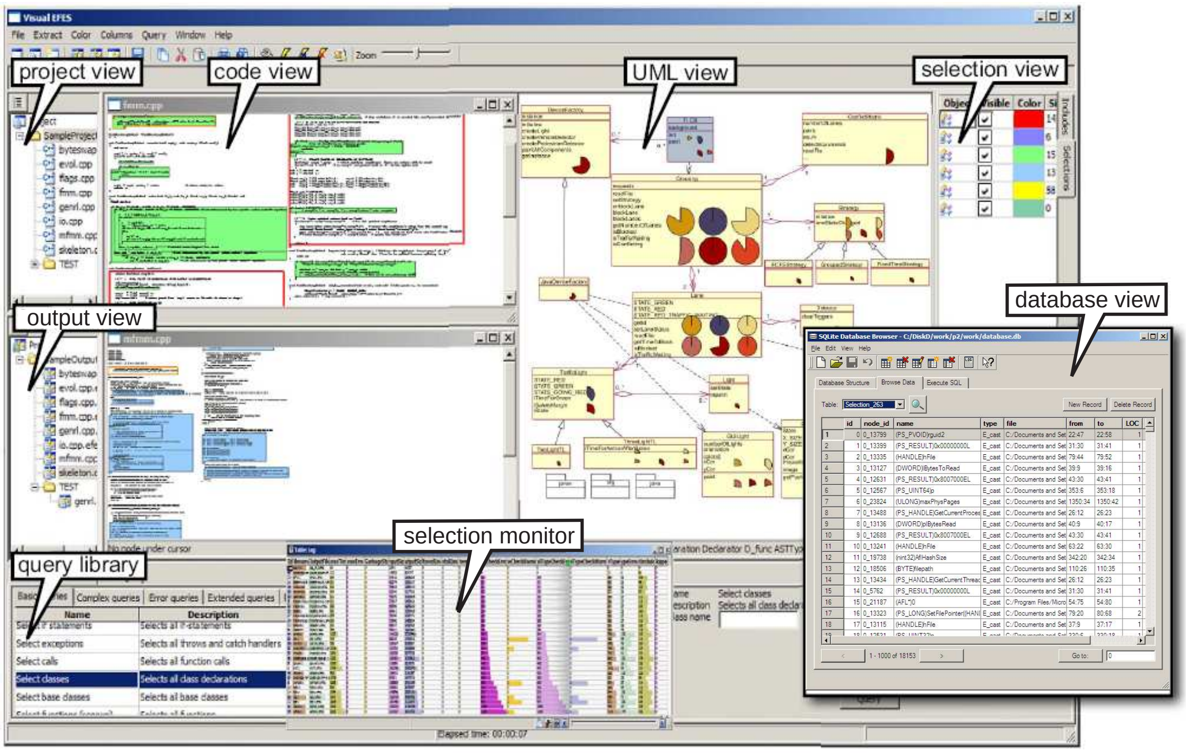

Figure 4. Overview of the SOLIDFX **Integrated Reverse Environment**

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, up to the point when one code line becomes a (colored) pixel line, thereby allowing one to overview larger amounts of code than using standard editors.

Queries can be applied in two ways, as follows. To perform a query, *e.g.* ”select all function definitions with n parameters”, one first 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. Secondly, one can right-click on any code element in the code view, exposing a menu with all queries which can take the clicked code element as input. Upon selection of a query, this is applied on the clicked code, and its result is 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*. The selection monitor is essentially a table having the selection elements as rows and their textual representation, code location, and various code metrics (if any) as columns. Some selections can have thousands of elements, *e.g.* the result of a ”select all functions” query. Understanding such a table can be difficult. To facilitate the creation of overviews, we use the well-known ’table lens’ technique [24]: 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 code metric values, so the entire table becomes a set of vertical graphs. Clicking on the table column headers sorts the respective columns on their values, thereby enabling one to quickly find those code elements having a minimal or maximal value of a given metric.

The *UML view* is a custom view showing UML class diagrams. The diagrams themselves are extracted from the fact database using queries which search for classes, inheritance relations, and associations. Associations 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 [25] 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 [26]. The combination of diagrams and metrics enables users to perform various types of code quality and modularity assessments, as shown further in Sec. 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 SQLite 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. In contrast, the built-in views are preferred in scenarios which heavily access the fact database at a fine-grained level.

4 Applications

We illustrate now the usage of the query and metrics engines added to SOLIDFX with several examples from a number of industrial projects[4]. The main characteristics of the applications discussed here are as follows:

- **Input:** A given C++ code base developed by a third-party team (*i.e.* not the persons doing the analysis).
- **Aim:** Assess a given quality attribute (*e.g.* modularity, maintainability, complexity) of a given C++ code base, and answer quality-related questions specific to each code base.
- **Method:** The code base is analyzed using our C++ parser; several queries and metrics are computed on the extracted fact database; the results are interactively examined using the SOLIDFX views and discussed with the project stakeholders.
- **Duration:** A typical analysis session takes a few hours from the initial code hand-over until the results are available. A complete code base assessment typically takes three to six such sessions, where increasingly refined questions and hypotheses are tested during the later sessions by means of specific queries on narrowed-down parts of the code base.

In the following, four such code assessments are described.

4.1 Finding Complexity Hot-Spots

In the first application, we examine the complexity of the wxWidgets code base, one of the most popular C++ GUI libraries having over 500 classes and 500 KLOC [27]. After extraction, we query all function definitions and compute several metrics on them: lines of code (LOC), comment lines ($CLOC$), McCabe's 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 (more files can be examined, we only selected these two for presentation clarity). 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 $f1$). Although complex, we 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 function, but the developers took extra care to comment it well.

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 this 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 border color). We see that $f1$ stands out as having both the fill and border in red (or dark gray in a monochrome printout), *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 (confirmed at close code inspection).

The exponential decrease of complexity shown by the colored $CYCLO$ bar-graph at the bottom of Fig. 5 is typical to the entire wxWidgets code base. Its interpretation is easy: there is a very small percentage of highly-complex code, the vast majority being of moderately low complexity. The highly-complex code is well documented. All in all, we conclude that wxWidgets has just a few complexity hot-spots, and these are well explained.

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, *i.e.* deciding which subsystems (if any) can be decoupled and ported in an incremental approach.

For this, we first extracted the static call graphs from the code, using a custom designed query that looks for function definitions and function calls, and links calls to the definitions using a technique which basically reproduces the working of a classical linker. This query also naturally and easily handles constructor, conversion-operator, and overloaded operator calls, since our C++ parser extracts and saves all this information in the AST. Using our query API, the complete code for the call graph extraction is under

[4] A video showing our tool is available at `www.solidsource.nl/video/SolidFX/SolidFX.html`

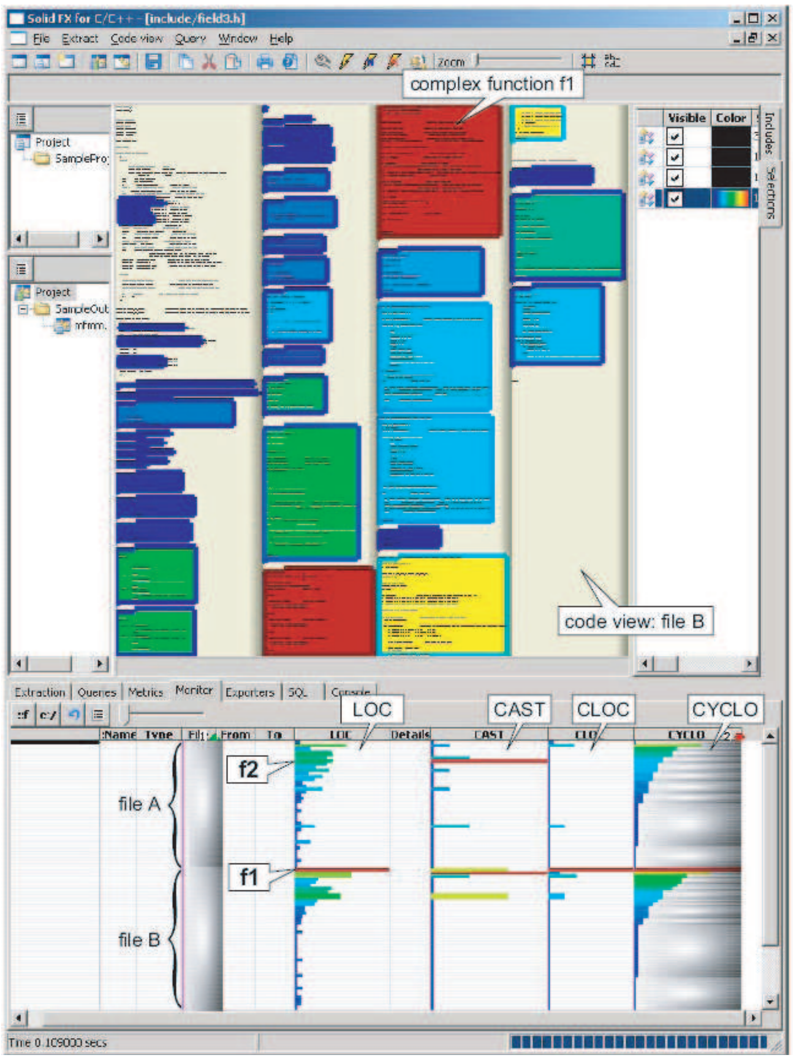
Solid FX for C/C++ - [include/field3.h]
File Extract Code view Query Window Help
complex function f1
code view: file B
Visible Color
Includes Selections
Extraction Queries Metrics Monitor Exporters SQL Console
LOC
CAST
CLOC
CYCLO
Name Type File From To LOC Details CAST CLO CYCLO
file A
file B
f2
f1
Time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

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

100 LOC of C++. 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 [28]. 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, as described in [28].

Figure 6 shows immediately a striking difference. The left subsystem shown is quite modular. We can easily discern the way its five subsystems (indicated by labels) call each other. Edge colors indicate the call direction: callers are red (medium-gray in a monochrome print-out), callees are blue (dark gray in a monochrome print-out). We immediately see, for example, that *libraries* is only called from *database* and that emphcore does not call *libraries*. This information can be directly used in the design of a phased porting plan. In contrast, the right subsystem, albeit of a similar size in terms of methods and classes, is far less modular. Here, we basically have two files which call each other in a highly complex way. There is little call structure to see, so little hope that one can easily split these files into smaller loosely coupled units and port these incrementally. Here, we used the edge color to show the call type: green indicates static calls, whereas blue shows virtual calls. This information is immediately available from a FunctionQuery where we ask for functions having the `virtual` attribute set. The blue edges appear to be somewhat bundled, so there is still some hope we can locate some interface classes (containing mainly virtual functions) in this way.

Overall, the stakeholders concluded that the left system is modular and envisaged a phased porting with relatively little difficulty. For the right system, the conclusion was that it is not modular, and the porting should be not attempted at this stage.

4.3 Maintainability Assessment

In the third application, we are interested to assess the maintainability of a C++ code base implementing an editor using OpenGL, wxWidgets, and the Standard Template Library (STL). 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. His managing architect was concerned about the code maintainability, for which this developer could not give a clear indication.

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 (functionally or otherwise). 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 number of methods. Inheritance relations are drawn as black (bold) lines, while associations are drawn as light-gray (thin) 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 suggests a good maintainability. Further, we see the heavy use of a few STL classes, mainly for the data model. This does not pose any maintenance problems, as it was agreed to use STL in the system implementation from the beginning, and STL is stable and well-documented software.

Atop the class icons, we visualized the computed LOC and $CYCLO$ metrics using colored bar-graphs. Long, red (medium gray) horizontal bars indicate high values. Thin, blue (dark gray) bars indicate low values. Within each class, the bar graphs are sorted from top to bottom in decreasing order of the $CYCLO$ metric. Looking at Fig. 7 top, we quickly discover 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 names of the most complex methods of the entire system, found of the class X. The most complex method has a McCabe value of 40, which is very large. Looking in detail at the code of X using a code view (Sec. 3.4), we could later see that it was indeed very complex. However, there is an essential observation that the UML view lets us perform: The diagram shows us also that class X is *not* referred to directly from outside the visualization plug-in. Plug-ins are optional in this system, so their maintainability is far less crucial than that of the system core. The lead developer recognized the complex class X 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 stay confined within the plug-in. Overall, the architect concluded that the entire system is in a satisfactory maintainability state, and recommended clean-up and refactoring work on the plug-in.

4.4 Change Propagation Assessment

We consider now the same editor code base as for the maintainability assessment (Sec. 4.3). During its development, its architect noticed that coding due to change propagation (*e.h.* modifying an interface or data field that is used frequently) consumed a higher-than-expected amount of time. In this fourth and last analysis, we want to assess whether our system is resilient to changes. In other words: would a change in the code of a class trigger lots of changes in other classes, due to data-dependencies?

Figure 8 shows a UML diagram extracted from source code, as explained earlier in Sec. 4.3. For each method, we now compute two new metrics: the number of variables read (*INPUTS*), respectively written (*OUTPUTS*). Metrics are sorted in decreasing order of *INPUTS*, and visualized with scaled bars, blue (dark gray in a monochrome printout) for *INPUTS* and purple (medium gray) for *OUTPUTS*. Both ranges of *INPUTS* and *OUTPUTS* are set to the same value, since the metrics have the same dimensionality.

We quickly see that there is no correlation between *INPUTS* and *OUTPUTS* values, but also discover some interesting outliers. The class marked A reads and writes a lot. This class is responsible for the rendering of UML model elements. Following the UML diagram, we discover it inherits from a *Visitor* interface. Looking at its method signatures, we understand that it accepts objects of UML Data Model types through its *Visitor* interface. A quick code browse of this class shows that the high read and write metrics are actually due to the $Visitor$ pattern implementation. Since this is a clean design pattern, we assess that the strong dependency of *UMLModelVisualizer* from the Data Model subsystem is a safe, acceptable one.

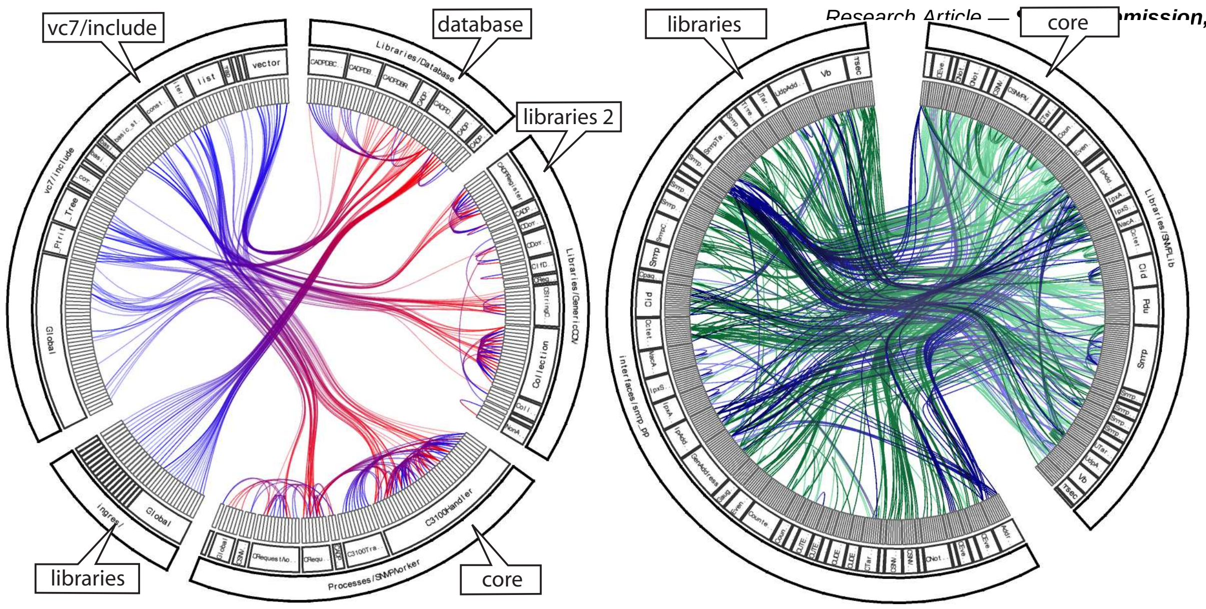

Figure 6. Call graph visualizations. Modular system (left) versus 'spaghetti code' (right)

We also discover the class B that reads a lot of data (high *INPUTS* metrics on most of its methods). Looking at its association relations (arrows on the UML diagram), we discover that this class has a *single* relation, which is actually an arrow (read) pointing to the *std::pair* class, which belongs to the STL library. Since STL can be considered as a very stable component, we conclude that our class B is also resilient to change.

Overall, our system's classes have a low number of external data-dependencies. For those few with numerous read and/or write dependencies as classes A and B), we could examine these dependencies, by following the diagram's relationships, and see that they point to stable components. All in all, we conclude that the system is stable with respect to change propagation.

5 Discussion

To assess the effectiveness of our proposed solution for static analysis of C++ code bases, we refer again to its three main ingredients introduced in Sec. 1.

5.1 Parsing

The extension of the tolerant heavyweight ELSA parser with preprocessor data, location information, error recovery, and filtering was essential for its success in an interactive click-to-query environment such as SOLIDFX. Integrating the preprocessor was needed since users want to analyze the code as they have written it, and not as the preprocessor expands it. Location information is needed to perform the code highlights (see *e.g.* Fig. 5) and click-to-query mechanism. Error recovery is needed since all of the four code bases considered here contained syntax errors and/or missing includes at the analysis time. Finally, filtering improved the query speed by one order of magnitude (Sec. 3.3.3).

It is interesting to consider whether all these features could be offered by, or added to, another C++ parser than ELSA. The set of mature available parsers for C++ is however extremely limited (Sec. 2). More importantly, even fewer of those are open source and/or modularly designed to easily incorporate additions. At the present moment, we do not know of any other C++ parser that could provide (with or without additions) all the functionalities required by our context.

5.2 Querying

The query engine, designed using the visitor pattern and the user-designed query-trees, was extremely effective in writing new queries. Once a basic set of around 40 queries was developed, subsequent queries were coded quickly in a matter of minutes, as a query takes on the average 40-50 XML lines. The query composition was used less than expected, one preferring simply to cut-paste-modify existing queries. This can be explained by the small code size of a query. Also, it is fair to say that, so far, the stakeholders did not design their own queries, but relied on one of the authors to do that, given that he was very fast in this task. The efficiency of the query system proved to be sufficient

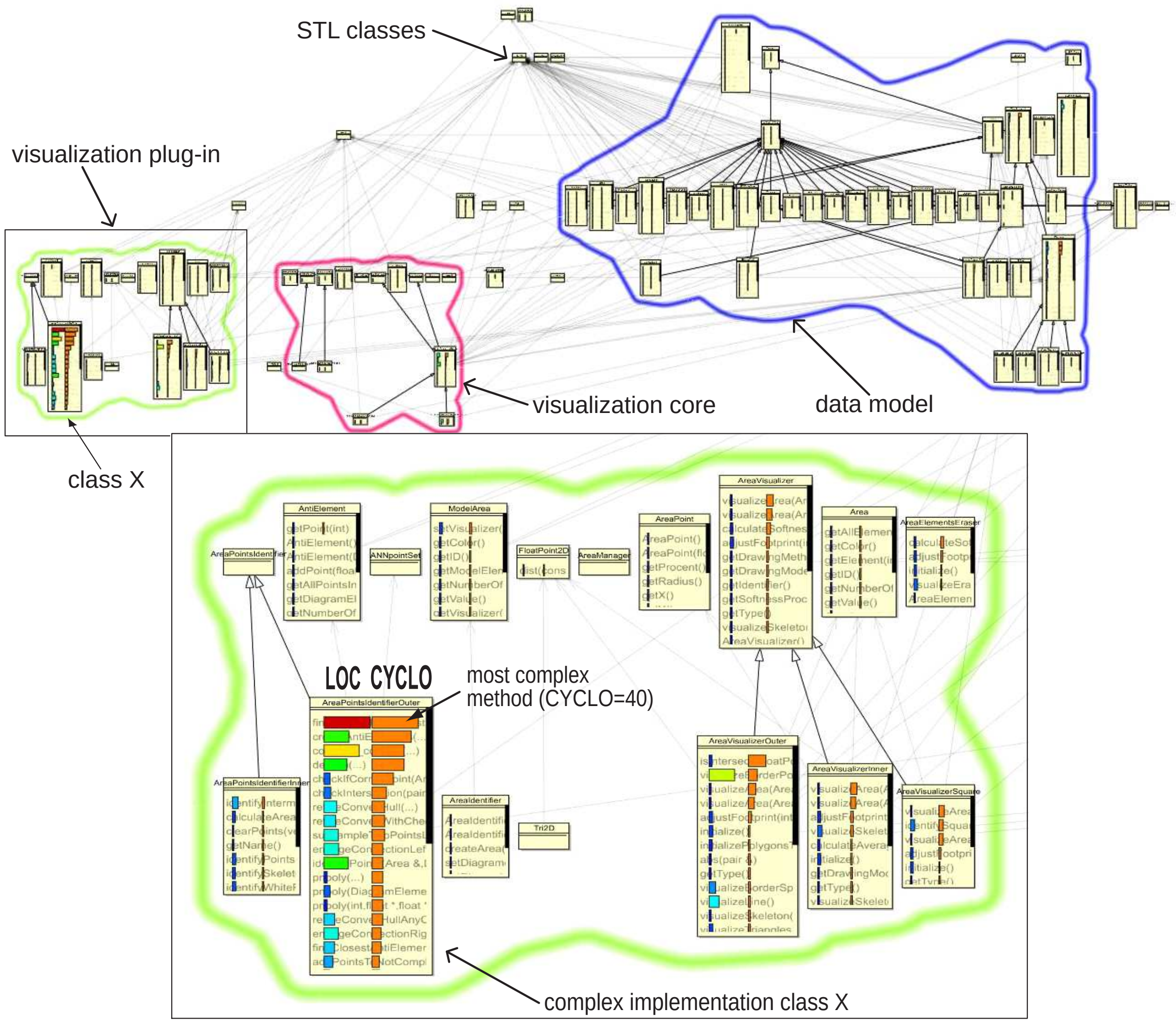

Figure 7. Maintainability assessment. Model-View-Controller architecture view (top). Zoomed-in view on the subsystem containing the most complex class (bottom)

for our interactive click-to query functionality.

The examples presented in this paper involve only relatively simple metrics: lines of code, lines of comments, cyclomatic complexity, number of read and written variables, and number of C casts. In turn, these involve several queries: select comments, control statements, local and global variables and function parameters, function call graphs, class inheritance relations, and C cast expressions. The selection of the examples and their subsequent queries and metrics was done on purpose to illustrate our engines on relatively simple scenarios that are relevant and apply to a large audience, and demonstrate the generic character of our solution. It is important to stress again that we can design more complex queries, metrics, and scenarios with the same ease as for the simpler cases. However, detailing such scenarios (and their code bases) would take considerable space and is beyond the scope of this paper. We plan to do this in an upcoming publication.

Although powerful, the visitor and query-tree combination is essentially a (flexible) pattern matching engine. As already noted in Sec. 3.3, more complex, context-dependent queries, or fuzzy queries, suchas needed to support the MISRA standard [29], have to be implemented manually based on the AST iterators which are part of our query API. However, since all these queries essentially subclass a single Query interface (Fig. 3), they are directly available to be composed with existing queries. As our experience progresses, we plan to work on higher-level refinements of the query API, such as detection of more complex design patterns, and possibly program slicing and dataflow analysis.

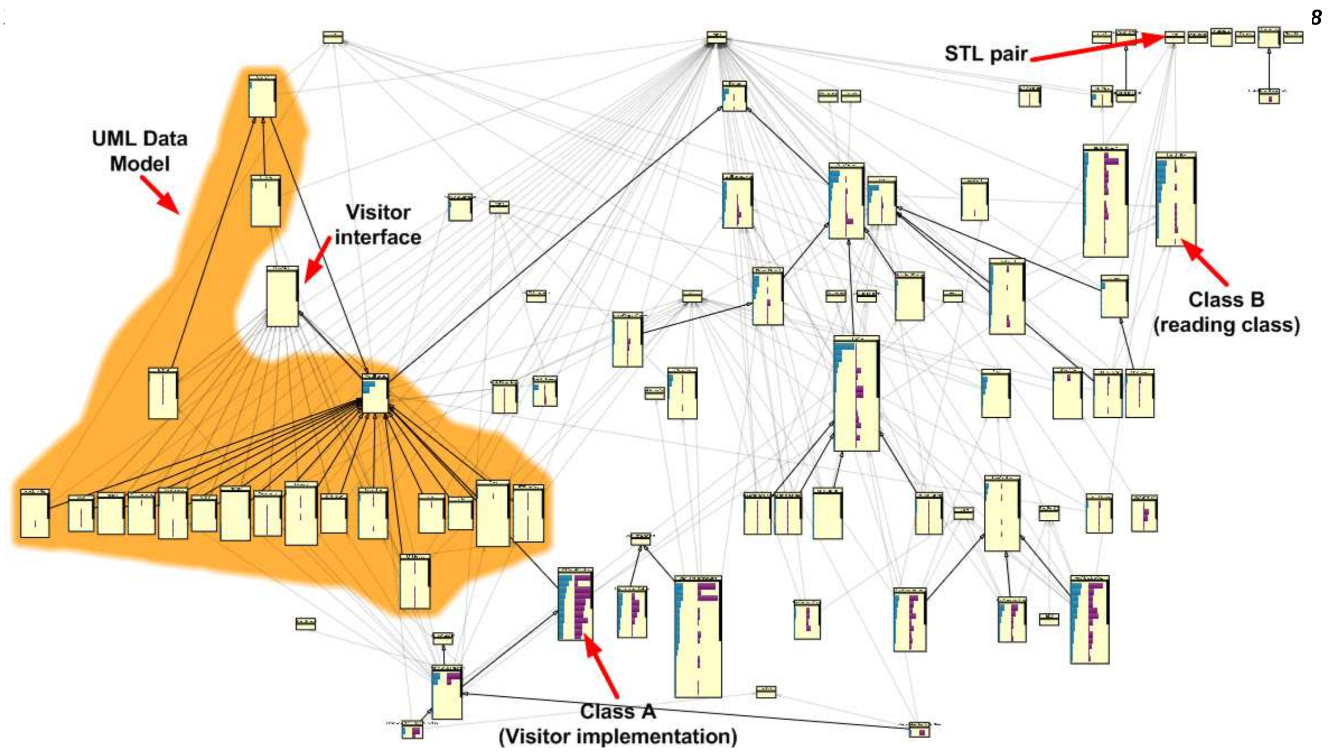

Figure 8. Change propagation analysis

5.3 Presentation

The basic views already existing in SOLIDFX, *i.e.* the code view, selection view, and UML view, seem so far to be sufficient to allow a simple but effective usage of the query and metric facilities solely via the GUI. The single largest request for a new presentation interface from users was to allow presenting relations and source code in a single view, *e.g* to show the declaration locations of various symbols directly in the code editor, at the locations where they are used. As we are not aware of the existence of such a technique, we are working in the direction of designing a new visualization to support this.

5.4 General remarks

All in all, we believe that the integration of our query-and-metrics engine in the SOLIDFX IRE largely improved the usefulness of this tool, which can now do full C++ parsing, custom analyses, and visualization, all in one environment. As noticed during the applications described here (and others), this integration considerably shortens the time needed from code-base acquisition to assessing specific questions related to its static analysis. We believe that this tight feature integration is paramount to the success. During numerous pilot projects, as early as [17], we observed that some of the greatest obstacles in the acceptance of a proposed static analysis tool or technique 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. Few stakeholders in the software industry are willing to invest the effort required in all the above, and few tools offer the three-element integration we propose here.

For the *extraction* phase, we also noticed that using the SOLIDFX IRE was not much more effective than using scripted command-line tools, which is often performed in batch mode [6, 7, 5, 10]. However, for the exploration phase, where new queries *and* presentations need to be specified and combined on-the-fly, the IRE and its tight tool integration were considerably more productive than using the same tools standalone, connected by small scripts and data files.

6 Acknowledgements

The work presented here on the design of SOLIDFXhas started in early 2005. As for most real-world software tools, feedback from a wide range of software engineers, including potential users and customers, is crucial for our work. We are truly grateful to the discussions fostered by prof. Derrick Kourie on our work on software visualization, his generous support that made it possible for us to present our work on visual C/C++ software analysis at three consec-

utive FASTAR/Espresso workshops (2005-2008), and his strong support that has enabled us to start several joint projects on static software analysis with several banking and insurance industry companies in the course of the year 2008.

7 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++. Our query framework involved the modifications of an existing heavyweight C++ parser to add preprocessor support, location information, parse error recovery, and output filtering, and the design and implementation of a new composable query API based on a pattern-matching visitor architecture. The integration of the modified parser and query API in the SOLIDFXintegrated reverse-engineering environment offers 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, our solution enables users to conduct reverse-engineering sessions with the same ease as software development is traditionally done in IDEs. We illustrated the application of our solution to four typical assessments involving static analysis of C++ code bases.

We are currently working on extending our C++ static query framework in several directions. Refined static information can be queried from the basic facts, such as control flow and data flow 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 our query system, such as checking for the MISRA C Standard [29], thereby making our framework more readily applicable in a number of industry use-cases.

REFERENCES

[1] B. Stroustrup. *The C++ Programming Language, 3^rd edition*. Addison-Wesley, 2000.

[2] M. L. Collard, H. H. Kagdi and J. I. Maletic. "An XML-Based Lightweight C++ Fact Extractor". In *Proc. IWPC*, pp. 134–143. IEEE Press, 2003.

[3] P. Mihancea, G. Ganea, I. Verebi, C. Marinescu and R. Marinescu. "McC and Mc#: Unified C++ and C# design facts extractors tools". In *Proc. SYNASC*, p. 101104. 2007.

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

[5] I. Baxter, C. Pidgeon and M. Mehlich. "DMS: Program Transformations for Practical Scalable Software Evolution". In *Proc. ICSE*, pp. 625–634. 2004.

[6] R. Ferenc, I. Siket, and T. Gyimóthy. "Extracting facts from open source software". In *Proc. ICSM*. 2004.

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

[8] S. McPeak. "Elkhound: A fast, practical GLR parser generator". Computer Science Division, Univ. of California, Berkeley. Tech. report UCB/CSD-2-1214, Dec. 2002.

[9] Edison Design Group. "The EDG C++ Front-End". 2008. www.edg.com.

[10] Y. Lin, R. C. Holt and A. J. Malton. "Completeness of a Fact Extractor". In *Proc. WCRE*, pp. 196–204. 2003.

[11] G. Lommerse, F. Nossin, L. Voinea and A. Telea. "The Visual Code Navigator: An Interactive Toolset for Source Code Investigation". In *Proc. InfoVis*, pp. 24–31. 2005.

[12] Abraxas Software. "CodeCheck for C and C++". 2008. www.abxsoft.com.

[13] Scientific Toolworks Inc. "*Understand* for C++". 2008. http://www.scitools.com.

[14] S. Eick, J. Steffen and E. Sumner. "Seesoft - A Tool for Visualizing Line Oriented Software Statistics". *IEEE Trans. Soft. Eng.*, vol. 18, no. 11, pp. 957–968, 1992.

[15] M. A. Storey, K. Wong and H. A. Müller. "How do program understanding tools affect how programmers understand programs?" *Science of Computer Programming*, vol. 36, no. 2, p. 183207, 2000.

[16] M. Lanza. "CodeCrawler - Polymetric Views in Action". In *Proc. ASE*, pp. 394–395. 2004.

[17] A. Telea. "An Open Architecture for Visual Reverse Engineering". In *Managing Corporate Information Systems Evolution and Maintenance (ch. 9)*, pp. 211–227. Idea Group Inc., 2004.

[18] S. Diehl. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[19] SolidSource BV. "SOLIDFX product information". 2008. www.solidsource.nl/products.

[20] F. Boerboom and A. Janssen. "Fact Extraction, Querying and Visualization of Large C++ Code Bases". 2006. MSc thesis, Eindhoven Univ. of Technology.

[21] B. Karlsson. *Beyond the C++ Standard Library - An Introduction to Boost*. Addison-Wesley Professional, 2005. See also www.boost.org.

[22] G. Knapen, B. Laguë, M. Dagenais and E. Merlo. "Parsing C++ despite missing declarations". In *Proc. IWPC*, pp. 114–122. 1999.

[23] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[24] A. Telea. “Combining extended table lens and treemap techniques for visualizing tabular data”. In *Proc. EuroVis*, p. 5158. 2006.

[25] AT & T. “GraphViz”. 2007. `www.graphviz.org`.

[26] M. Termeer, C. Lange, A. Telea and M. Chaudron. “Visual exploration of combined architectural and metric information”. In *Proc. VISSOFT*, pp. 21–26. 2005.

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

[28] D. Holten. “Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data”. In *Proc. InfoVis*, pp. 741–748. 2006.

[29] MISRA Association. “Guidelines for the use of the C language in critical systems”. 2008. `www.misra-c2.com`.