Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 233 (2009) 143–159

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Framework for Reverse Engineering Large C++ Code Bases

Alexandru Telea[1] Heorhiy Byelas[2]

Institute for Mathematics and Computing Science
University of Groningen, the Netherlands

Lucian Voinea[3]

SolidSource BV
Eindhoven, the Netherlands

Abstract

When assessing the quality and maintainability of large C++ code bases, tools are needed for extracting several facts from the source code, such as: architecture, structure, code smells, and quality metrics. Moreover, these facts should be presented in such ways so that one can correlate them and find outliers and anomalies. We present SolidFX, an integrated reverse-engineering environment (IRE) for C and C++. SolidFX was specifically designed to support code parsing, fact extraction, metric computation, and interactive visual analysis of the results in much the same way IDEs and design tools offer for the forward engineering pipeline. In the design of SolidFX, we adapted and extended several existing code analysis and data visualization techniques to render them scalable for handling code bases of millions of lines. In this paper, we detail several design decisions taken to construct SolidFX. We also illustrate the application of our tool and our lessons learnt in using it in several types of analyses of real-world industrial code bases, including maintainability and modularity assessments, detection of coding patterns, and complexity analyses.

Keywords: reverse engineering, parsing, C++, software visualization

1 Introduction

In the last years, several tools have emerged to support program understanding, software maintenance, reverse engineering, and reengineering activities. A large part of such tools extract their information mainly from the source code via static analysis. This includes a set of operations ranging from code parsing and fact extraction, fact aggregation and querying, up to interactive presentation.

[1] Email: `a.c.telea@rug.nl`

[2] Email: `h.v.byelas@rug.nl`

[3] Email: `lucian.voinea@solidsource.nl`

1571-0661/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2009.02.066

Several requirements are to be met by static analysis tools to be accepted and used by the software industry. First, the fact extractors should be able to deliver detailed low-level information, *e.g.* complete syntax trees, from possibly incorrect and incomplete code bases of millions of lines-of-code (LOC). Next, subsequent tools are needed to query the subsets of facts of interest from the raw data, *e.g.* select all code fragments which comply with a given quality or safety standard. Of particular interest is the recovery of high-level design information, *e.g.* class diagrams, from source code. Computing quality metrics and code smells are important instruments in this process. Third, interactive presentation and exploration mechanisms should be provided to drive the entire understanding process. Finally, all above tools should be tightly integrated, scalable, and simple to use and learn.

For the C++ language, there are few static analysis and reverse engineering toolsets which meet all above criteria. Robust static analysis of industry-size C++ code bases is particularly hard, given the language complexity, its several dialects, the presence of mixed C and C++ code, and the use of a preprocessor. Moreover, although several C and C++ static analyzers exist, few come integrated in a framework with query, metrics, and interactive visualization tools which make them truly scalable and easy to use by software engineers.

In this paper, we present our experience in the architecting of SolidFX, an Integrated Reverse-Engineering environment for C and C++. SolidFX is a commercial product which provides full analysis of industry-size C and C++ code bases, customizable query and metric engines, and extraction of UML class diagrams from source code [22]. All analysis operations supported by SolidFX are available via an interactive user interface which uses several novel visualization techniques to achieve scalability. SolidFX was designed to cover a wide range of analyses, ranging from local variable level up to system architecture, and also as an open environment which allows integration of third-party analyses via plug-ins. Overall, SolidFX 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 tool architecture of SolidFX and details its main components: the parser, the query and metric engine, and the data views which SolidFX adds atop of fact extraction and analysis engines. 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

Related work in the area of interactive reverse engineering can be grouped in two categories: extraction and analysis of facts from source code, and visual presentation of the results.

In the first category, 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 [6], Sniff+, GCCXML, MCC [18], and several custom analyzers constructed using the ANTLR parser-generator [20]. 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 (visual) analyses, 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 [2], Columbus [9], ASF+SDF [28], Elsa [17], and CPPX [15]. 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 [16]. 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.

The output of static analyzers, mostly produced by runnign batches, can be fed to a range of *visualization* tools. Many such tools exist, ranging from line-level, detail visualizations such as SeeSoft [8] up to architecture visualizations which combine structure and attribute presentation, *e.g.* Rigi [23], CodeCrawler [13], or SoftVision [24]. An extensive overview of software visualization techniques is provided by Diehl in [7].

3 IRE Architecture

3.1 Environment Requirements

Our quest to design an Integrated Reverse-engineering Environment (IRE) that offers a similar flexibility and ease-of-use to the reverse-engineering process as IDEs do to the forward engineering started with an early toolset version, called the Visual Code Navigator (VCN) [16]. Initial results obtained with the VCN during several projects involving commercial, open-source, and academic C++ code, as well as our own experience in using Columbus for in-house fact extraction, led us to the following main design considerations and requirements:

- a *tolerant* extractor is strongly preferred to an incomplete one. In both early and late reverse-engineering and analysis phases, users would want to be able to analyze code that does not compile for a variety of reasons, and would not accept a strict extractor that fails on preprocessor, syntax or type errors.

- a *heavyweight* extractor is preferred to a lightweight one for several reasons. First, users need to design custom, project or task-specific, queries and code metrics that uses a wide range of fact types, so we need to be able to produce these. Second, in order to pose queries on-the-fly on source code and also present the analysis results in code-level visualizations, we need fine-grained information such as the location, scoping, and types of each identifier in the source code. Such a level of detail requires a heavyweighr extractor.
- a tight *integration* of the extraction, analysis, and visualization tools in a single, coherent, easy-to-use environment is essential for acceptance. Many users would not accept a steep learning curve for a complex tool, while they are ready to experiment with a point-and-click, error-tolerant interface.

Constructing a scalable solution that complies with all these requirements is quite difficult. Although several heavyweight C++ extractors exist (see Section 2), such tools typically work in batch mode and do not expose a fine-grained API that allows their integration into the complex control structure required by an interactive environment. Also, implementing an open set of custom queries on the extracted facts in an efficient way is quite difficult. Finally, most such extractors are closed source, making their modification impossible.

For these reasons, we decided to base our IRE design on a heavyweight C++ fact extractor of our own construction. The entire IRE consists of this fact extractor, a query and metric computation engine, and several data views. All these components communicate via a central *fact database* (Figure 1). In the following sections, we describe each of these components.

3.2 Fact Database

The fact database contains both the raw facts, produced by the extractor directly from the source code, as well as several derived facts, produced by the query and metric engines during the analysis and reverse engineering process (Sec. 3.4). The database acts as a central repository, read and written by the various components in the IRE (Fig. 1).

A database instance contains facts extracted from the analysis of a given *project*. A project is much like a makefile, *i.e.* contains a set of source files, include paths, `#defines`, and language dialect settings. Project descriptions can be automatically created by parsing either makefiles or Visual C++ XML project files, with a technique quite similar to the so-called compiler wrapping described for the Columbus extractor [9]. A project is analyzed one source file (translation unit) at a time by the fact extractor (Sec. 3.3), which saves information on four kinds of data elements (lexical, syntax, type, and preprocessor) in the database. Each data element is assigned a unique id. The database is structured as a set of binary files, one per translation unit, and a set of has maps which allow referring to the elements by id within each file.

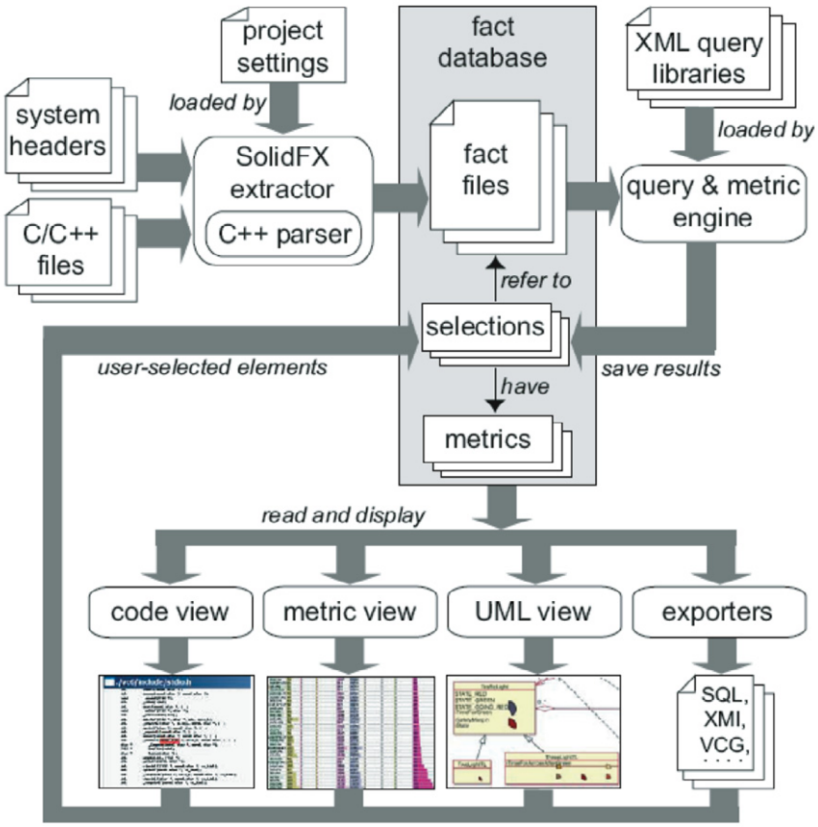

Fig. 1. Dataflow architecture of SolidFX

The IRE components communicate with each other by lightweight sets of fact ids, called *selections*, which resemble the table views in SQL databases. The database creation, done during parsing the source code, is by far the most consuming time of the static analysis. After this process is completed, queries and visualizations only modify selections, a process which can be done at interactive rates (Sections 3.4,3.5)

3.3 C++ Parsing

As outlined in Section 3.1, our SolidFX environment uses a C and C++ heavy-weight analyzer of own construction. We based this analyzer on Elsa, an existing C++ parser designed using a GLR grammar [17]. 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 several features required by an analyzer used in an interactive environment context (Section 3.1. Its most important limitations are as follows. 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 3.5), are missing. Second, error recovery lacks, so incorrect code causes parse errors. Third, the output cannot be filtered or queried for specific facts.

We have extended Elsa to eliminate all these limitations [3]. Our 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.

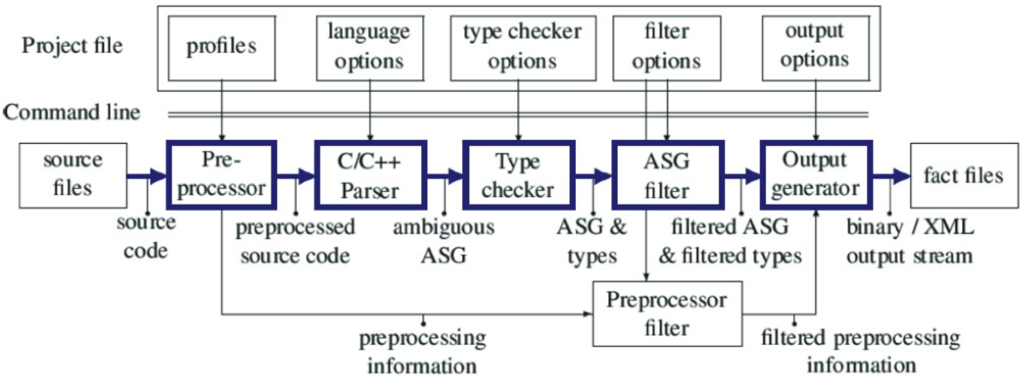

Fig. 2. Architecture of the SolidFX C++ parser (main components shown in bold)

The parser reads the token stream from the preprocessor as it performs reductions and builds the AST. We use here the Elsa parser, which we extended 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 declaration or function body containing the error was not present in the input. This approach required adding only six extra grammar rules to the original C++ GLR grammar of Elsa. Our approach, where error-handling grammar rules get activated on demand, resembles the hybrid parsing strategy suggested by [12]. 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, we believe that our design balances well implementation simplicity with a good granularity of error recovery.

The parsing phase, augmented by error recovery, is followed by the original AST dismbiguation and type-checking implemented by the Elsa parser. After this step, 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 3.2). In other words, the filtering phase eliminates all AST nodes which are present in the included *headers* but are not referred to by AST nodes contained in the analyzed *sources*. This eliminates 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 [4]. Finally, the filtered output is written to file using a custom binary format.

All in all, the several design choices made during the fact extractor implementation, *i.e.* using the ELSA parser which has a highly optimized core written in C; 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 an efficient heavyweight parser. SOLIDFX is roughly three to six times, on the average, faster than COLUMBUS, one of the fastest heavyweight C++ parsers that we could test, and also scales well to analyze projects of millions of lines of code [3]. 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.

3.4 Query and Metrics Engine

The second main component of the IRE is a customizable *query and metrics engine*. A query implements the function

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

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 implemented as a C++ class library which implements several specializations of the above query interface q, as follows. Each syntax-node T in the C++ GLR grammar of our parser has several children $c(T)$ and several data attributes $a(T)$. For example, a method-node has a signature, a function-body child, and a 'virtual' boolean attribute. For each such node T, we generate a query-node specialized to check for the values of the attributes $a(T)$ and/or call the query-children of $c(T)$. For example, a function query-node whose 'virtual' attribute is set to 'true' will only match AST nodes for functions declared virtual. Query-nodes can be assembled in query-trees, yielding composite queries. Query trees can be further composed using logical operators, such as AND, OR, NOT. A given query tree q is applied on a given 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.

The above 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". Query trees can be serialized into an XML-based file format, thereby allowing users to

[4] 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

design custom queries and query libraries simply by editing XML files, *i.e.* without recompiling the query engine. Several examples of queries and their applications are presented next in Section 4.

A careful query engine design makes applying queries on large fact databases is very efficient. First, since the parsed AST nodes are indexed in the binary database by their ids, visiting large query trees can be done very efficiently directly on disk, by visiting their respective index offsets. The results of a query are selections, *i.e.* lightweight index-sets. Combined with early query termination, this lets us query fact databases of millions of AST nodes in subsecond time, once the code is parsed.

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

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

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 [14] can be implemented in this way.

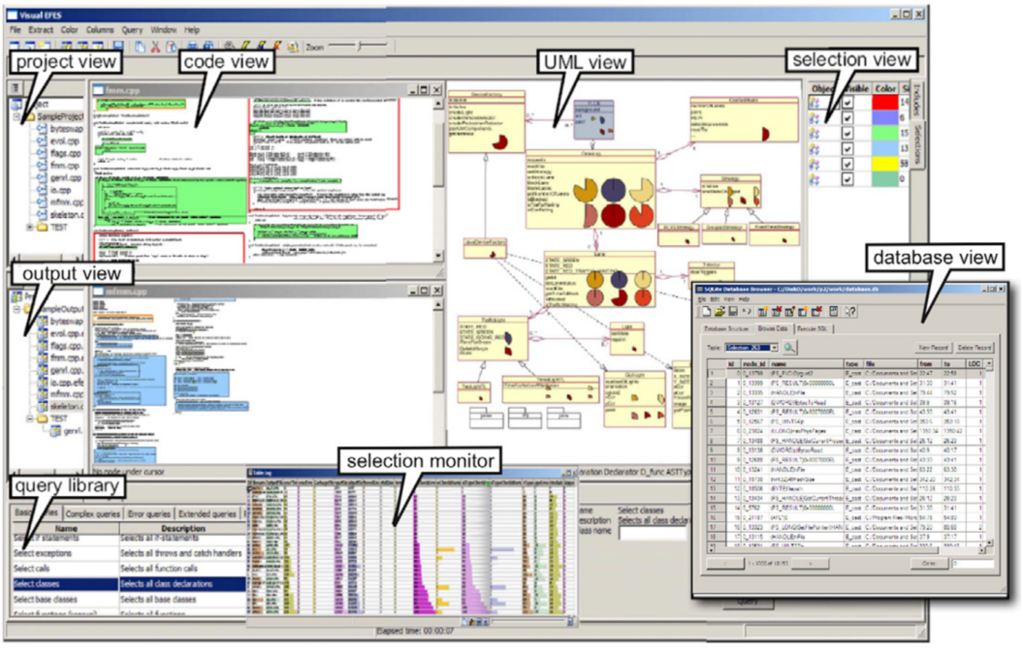

Fig. 3. Overview of the SolidFX Integrated Reverse Environment

3.5 Data Views

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

Figure 3 shows the main data views we developed for our IRE. The *project view* lets users set up a fact extraction project, much like one sets up a build project in Visual Studio or Eclipse. The fact database files created by the C++ parser are shown in the *output view.* The *selection view* shows all selections available in the fact database. The appearance of each selection can be customized here, *i.e.* one can specify if it is visible, and if so, how to color its elements (as discussed next) [5]. The *query view* shows all available queries in the XML-based query library (Section 3.4). 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 our example, in the query GUI, and clicks on the selection on which to perform the query in the selection view. The query's output gets automatically added as a new selection to the selection view. A separate view, called the *selection monitor*, shows all the code elements in the currently active selection in the selection view, together with their computed metrics. The metrics are displayed using a combination of text and colored bar graphs based on the so-called 'table lens' technique [25]. When zoomed in, the table lens behaves like a usual Excel table. When zoomed out, each row is reduced to a pixel row colored and scaled to show the data values, and thereby the entire table is reduced to a set of vertical graphs.

The *code views* show the actual source code in the desired files. The code in the visible selections is highlighted in the respective selections' colors, thereby enabling one to spot the occurrence of particular events. Constructing such highlights is easy, since we have the exact (row,column) locations of every AST node and preprocessor directive from the parsing phase (Sec. 3.3). Just as the selection monitor, 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 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 [1] or a custom graph layout library developed by us. 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 [27]. The combination of diagrams and metrics is a powerful instrument for performing 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. 3 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

[5] We recommend viewing this document in full color

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 samples from its actual usage within a number of industrial projects[6]. 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[7]. 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.

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 [21]. 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 4 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. 4 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 $CLOC$).

[6] A video showing SolidFX in use is also available at www.solidsource.nl/video/SolidFX/SolidFX.html
[7] The only exception to this is the wxWidgets code base

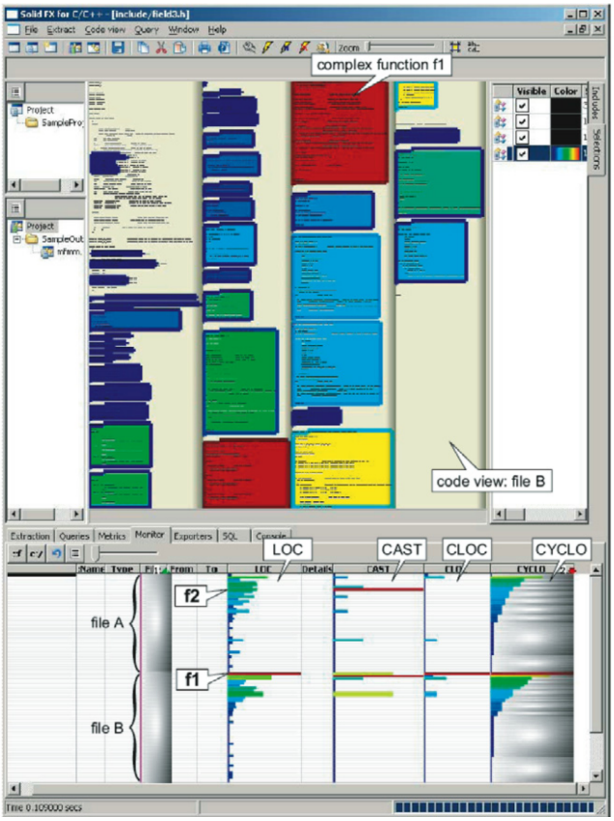

Fig. 4. Finding complexity hot-spots in the wxWidgets code base

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 [10]. 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. 5). Call relations are drawn as splines, bundled to indicate relations emerging from, or going to, the same hierarchy ancestor.

The subsystem shown in Fig. 5 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. 5 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 hope we can locate some interface classes (containing mainly virtual functions) in this way.

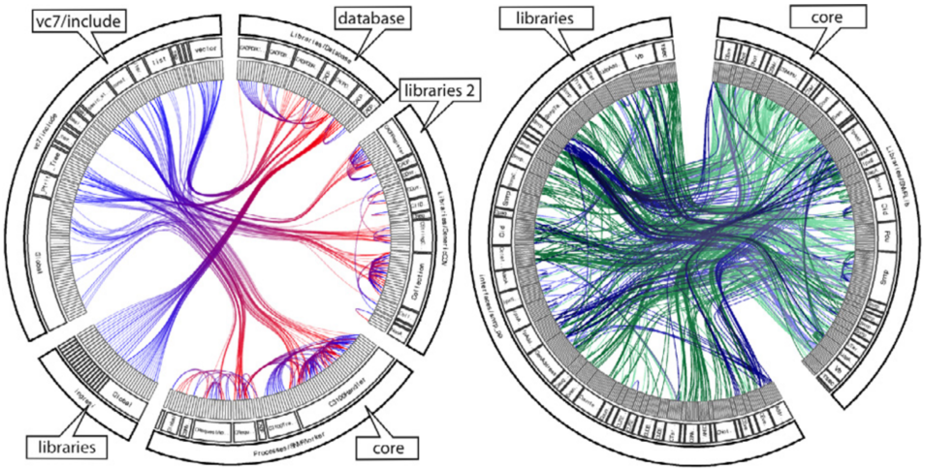

Fig. 5. Call graph visualizations. Modular system (left) versus 'spaghetti code' (right)

4.3 Maintainability Assessment

In the third 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 four persons [26]. 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 6 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 subsystems themselves are visualized by surrounding their contained classes by a fuzzy-shaped contour, using the so-called *areas of interest* technique, described in detail in [4]. These areas of interest are not overlapping, which shows that the considered 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. 6 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 6 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.

4.4 Analysis of a Copy-and-Paster Pattern

The presented examples so far have shown the usage of our IRE in performing high-level, global assessments of a code base. However, SolidFXcan also be used in more fine-grained analyses, as illustrated by this fourth and last application. Here, we study the same UML editor which was the subject of the maintainability analysis in Sec. 4.3. We now zoom in on a subset of the graphics rendering subsystem's classes, using the UML view. The rendering subsystem is indicated by the orange contour (Fig. 7) using the areas-of-interest visualization technique, which has already been introduced in Sec. 4.3. We display the LOC and COM metrics, and sort methods by LOC - hence, the top methods in the class icons are the largest, whereas the bottom-most ones are the smallest. Within the rendering subsystem, we see a smaller set of classes related by inheritance and which also have very similar metric patterns, as indicated by the lengths of the metric bars. These are marked by the dark filled area-of-interest in Fig. 7. Given the inheritance relations, these appear to be a three-level class hierarchy rooted at a class R.

Why would these classes exhibit such similar metric patterns? This was intriguing at first sight, so we started to browse the class and method names, using the

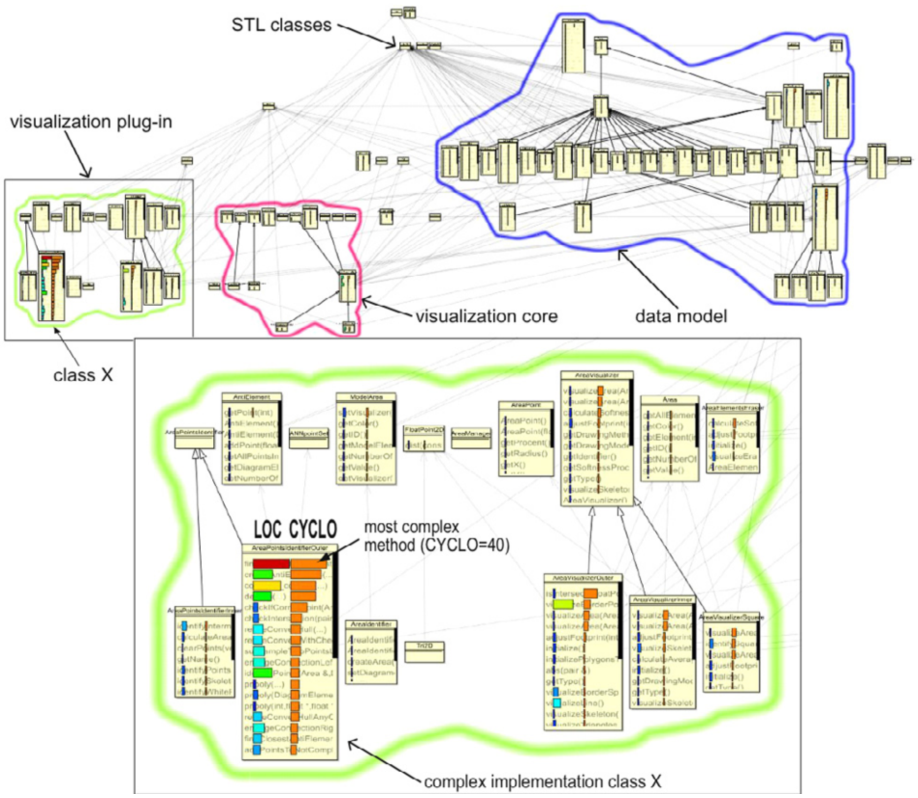

Fig. 6. Maintainability assessment. Model-View-Controller architecture view (top). Zoomed-in view on the subsystem containing the most complex class (bottom)

tooltip facilities of the UML view, to get more insight. After a while, an explanation emerged: these classes implement the different metric glyphs supported by the studied UML editor (*e.g.* pie chart, flat-colored bar, rainbow-colored bar, 3D bar, etc). Specically, these classes all implement the `MetricGlyph` interface (class R in Fig. 7) which has only a few methods, *e.g.* `draw()`. The reason why all the subclasses of R have such similar metrics became apparent once we looked at their actual code in the Code view: They exhibited nearly identical code fragments with the base class R. We discussed this finding with the developer responsible for these classes (who was not involved in the assessment described here) and he informed us that these classes were coded by copy-and-pasting the code from class R and making some small local modifications, mainly in the `draw()` method.

This analysis shows an interesting, collateral usage of the metric visualizations: Together with relations (*e.g.* inheritance), metric visualizations can be used as a first indicator of replicated code, attracting the user's attention to potential copy-and-paste events. Detecting such events is interesting in a number of maintenance scenarios, *e.g.* debugging, refactoring, migration, porting, and documentation. The usefulness of our integrated (IRE) solution becomes apparent, as the finding documented here could not have been possible withouth a *tight* integration of interactive metrics-and-UML visualization, complemented by the detailed code views.

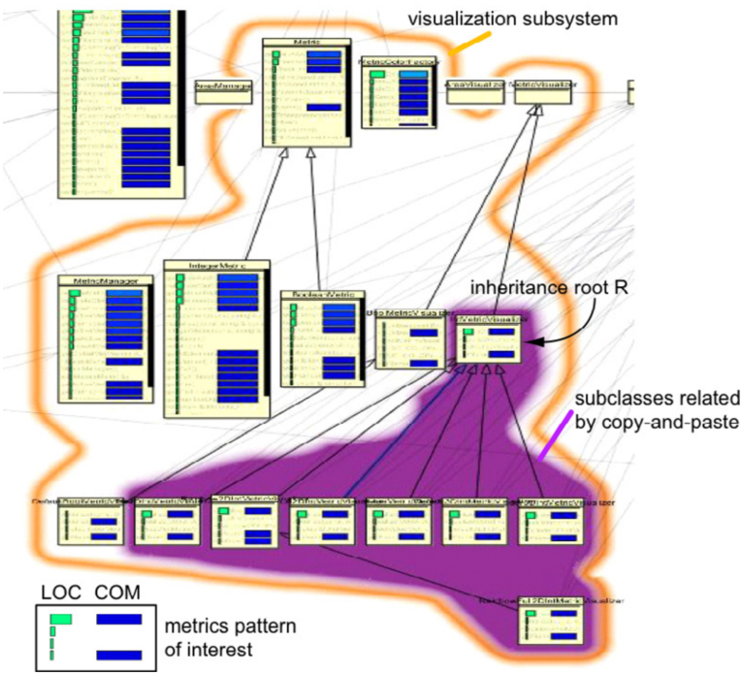
visualization subsystem
inheritance root R
subclasses related by copy-and-paste
LOC COM
metrics pattern of interest

Fig. 7. Identification of a copy-and-paste pattern in a class hierarchy

5 Discussion

SolidFX is a mature product, able to handle code bases of millions of LOC. During numerous pilot projects, as early as [24], we noticed that the greatest obstacle in the acceptance of the proposed visual analysis techniques was 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 only scripting) in order to use a toolset. The relative high success of SolidFX in several recent projects seems to be due to the high integration of its functions, presented under a uniform interface, and the possibility to execute complex analyses with minimal (or no) programming. Using the IRE was not much more effective than using scripted command-line tools for the extraction phase. However, for the exploration phase, the IRE and its tight tool integration were more productive than using the same tools standalone, connected by little scripts and data files. Reverse-engineering and code assessment scenarios are by nature iterative and exploratory, so they map perfectly to repeated selection, query, and interactive visualization operations.

It is tempting to consider the extension of our IRE solution to existing IDE tools, such as Eclipse or Visual Studio. The main problem, in both cases, regards the availability of an efficient heavyweight parser which offers also fine-grained access to its fact database. A recent step in this direction is CDT, an open C/C++ development toolkit for Eclipse [5]. CDT resembles the goal, and loosely has the same architecture as SolidFX, albeit in a much earlier consolidation phase. Visual Studio has the right type of parser infrastructure support built-in, so would be an excellent candidate. However, at the present date, the parser API is not sufficiently open(ed) to offer the type of fine-grained access necessary to provide the functionality discussed here.

6 Conclusions

We have presented the design of SolidFX, an Integrated Reverse Engineering environment for C and C++. SolidFX combines a heavyweight tolerant C++ parser with a fact database offering fine-grained access, an open query and metric engine, and several scalable visualizations that combine code, metrics, and diagrams. Due to the high integration of these tools, SolidFX enables users to conduct reverse-engineering sessions 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 extracted 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, such as checking for the MISRA C Standard [19], thereby making the use of SolidFX even more productive and easy.

Acknowledgement

We are grateful to the anonymous reviewers which supported us with comments on improving the selection of application scenarios. Part of the research presented here has been funded by the ITEA Trust4All European research project on component-oriented software architectures [11].

References

[1] AT & T. GraphViz. 2007. www.graphviz.org.
[2] I. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *Proc. ICSE*, pages 625–634, 2004.
[3] F. Boerboom and A. Janssen. Fact extraction, querying and visualization of large c++ code bases. 2006. MSc thesis, Eindhoven Univ. of Technology.
[4] H. Byelas and A. Telea. Visualization of areas of interest in software architecture diagrams. In *Proc. SoftVis*, pages 20–28. ACM Press, 2006.
[5] CDT Team. C/C++ development toolkit. 2007. www.eclipse.org/cdt.
[6] M. L. Collard, H. H. Kagdi, and J. I. Maletic. An XML-based lightweight C++ fact extractor. In *Proc. IWPC*, pages 134–143. IEEE Press, 2003.
[7] S. Diehl. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software.* Springer, 2007.
[8] S. Eick, J. Steffen, and E. Sumner. Seesoft - a tool for visualizing line oriented software statistics. *IEEE Trans. Soft. Eng.*, 18(11):957–968, 1992.
[9] R. Ferenc, I. Siket, , and T. Gyimóthy. Extracting facts from open source software. In *Proc. ICSM*, 2004.
[10] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. In *Proc. InfoVis*, pages 741–748, 2006.
[11] ITEA. Trust4All research project. In *www.win.tue.nl/trust4all*, 2007.
[12] G. Knapen, B. Laguë, M. Dagenais, and E. Merlo. Parsing C++ despite missing declarations. In *Proc. IWPC*, pages 114–122, 1999.
[13] M. Lanza. CodeCrawler - polymetric views in action. In *Proc. ASE*, pages 394–395, 2004.
[14] 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.
[15] Y. Lin, R. C. Holt, and A. J. Malton. Completeness of a fact extractor. In *Proc. WCRE*, pages 196–204, 2003.

[16] G. Lommerse, F. Nossin, L. Voinea, and A. Telea. The visual code navigator: An interactive toolset for source code investigation. In *Proc. InfoVis*, pages 24–31, 2005.
[17] S. McPeak. Elkhound: A fast, practical glr parser generator. Computer Science Division, Univ. of California, Berkeley. Tech. report UCB/CSD-2-1214, Dec. 2002.
[18] 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*, page 101104, 2007.
[19] MISRA Association. Guidelines for the use of the C language in critical systems. 2008. www.misra-c2.com.
[20] T. Parr and R. Quong. ANTLR: A predicated-LL(k) parser generator. *Software - Practice and Experience*, 25(7):789–810, 1995.
[21] J. Smart, K. Hock, and S. Csomor. *Cross-Platform GUI Programming with wxWidgets*. Prentice Hall, 2005.
[22] SolidSource BV. SolidFX product information. 2008. www.solidsource.nl/products.
[23] M. A. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2):183207, 2000.
[24] A. Telea. An open architecture for visual reverse engineering. In *Managing Corporate Information Systems Evolution and Maintenance (ch. 9)*, pages 211–227. Idea Group Inc., 2004.
[25] A. Telea. Combining extended table lens and treemap techniques for visualizing tabular data. In *Proc. EuroVis*, page 5158, 2006.
[26] A. Telea, M. Termeer, C. Lange, and H. Byelas. AreaView: An editor combining uml diagrams and metrics. In *www.win.tue.nl/~alext/ARCHIVIEW*, 2008.
[27] M. Termeer, C. Lange, A. Telea, and M. Chaudron. Visual exploration of combined architectural and metric information. In *Proc. VISSOFT*, pages 21–26, 2005.
[28] M. van den Brand, P. Klint, and C. Verhoef. Reengineering needs generic programming language technology. *ACM SIGPLAN Notices*, 32(2):54–61, 1997.