SOLIDFX: An Integrated Reverse Engineering Environment for C++

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

Lucian Voinea
SolidSource BV
Eindhoven, Netherlands
lucian.voinea@solidsource.nl

Abstract

Many C++ extractors exist that produce syntax trees, call graphs, and metrics from C++ code, yet few offer integrated querying, navigation, and visualization of source-code-level facts to the end-user. We present an interactive reverse engineering environment which supports reverse-engineering tasks on C/C++ code, e.g. *set up the extraction process, apply user-written queries on the extracted facts, and visualize query results, much like classical forward-engineering IDEs do. We illustrate our environment with several examples of reverse-engineering analyses.*

1. Introduction

Fact extraction from source code is a crucial step in static code analysis. To be effective in scenarios ranging from architecture extraction to code transformation and quality control, a fact extractor tool must be able to extract both high-level facts (*e.g.* call and inheritance graphs) and low-level (*e.g.* syntax trees) facts from often incorrect and incomplete millions of lines of code (LOC). Also, to be practical, fact extractors should be tightly integrated with query, analysis, and presentation tools, and simple to use.

For C++, few extractors meet these requirements. Even fewer come withing a framework which makes their use simple and effective for non-experts. We present here an Integrated Reverse-engineering Environment (IRE) for C++, which provides seamless integration of code analysis (parsing, fact extraction, metric and query computations) and presentation (source-code, metric, and relational data interactive visualizations). Overall, our IRE offers to reverse engineering users the look-and-feel of Integrated Development Environments (IDEs) such as Visual C++ or Eclipse.

2. Environment Architecture

Our integrated reverse-engineering environment (IRE) connects a parser, a query system, and several views (Fig. 1) via *selections*. These are sets of ASG node-IDs, enriched with metrics stored as (key,value) pairs along the nodes. The ASGs, selections, and metrics of all input source files create our complete *fact database*. We next discuss these tools.

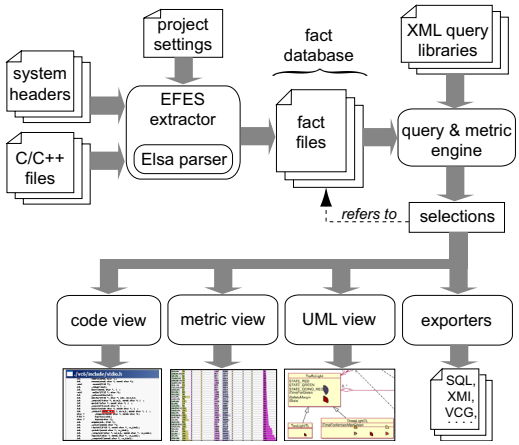

Figure 1. Architecture of the IRE

2.1. C++ Fact Extractor

We parse C and C++ using our own SOLIDFX fact extractor, built atop the ELSA parser [2]. ELSA's modular design cleanly separates C++ parsing (done using a Generalize Left-Reduce (GLR) grammar) from type-checking and disambiguation [5]. ELSA's speed is close to the gcc compiler. In SOLIDFX, we extend ELSA to handle parse errors by skipping from parsing the deepest scope where these occur. SOLIDFX can parse complex code bases such as STL, the Windows headers, or the Boost library. SOLIDFX outputs full Annotated Syntax Graphs (ASGs), complete with preprocessor data and line numbers. The output is detailed enough to fully reconstruct the input code from it.

2.2. Fact Querying and Metrics

The ASG of a 100 KLOC code base can contain over one million nodes. On such data, end-users need to ask both de-

tailed, low-level questions *e.g.* "show all calls of a default constructor whose body throws an exception called `X`", and high-level questions, *e.g.* "show the system's complete call graph". We provide an open *query system* that can answer both types of questions. For each of the 170 syntax node-types *T* in the GLR C++ grammar, we generate a query-node, containing children query-nodes for *T*'s non-terminal children and data properties for *T*'s terminal children. For instance, the `Function` query has a property *name* for the function's name, and two query-children *body* and *signature* for the function's body and signature. Queries can be assembled in query-trees. We also provide logical (*e.g. AND*, *OR*) and closure operators. To do lexical queries, properties can be set to regular expressions. For example, to find all functions whose name begins with "Foo" and have a parameter called "Bar", we set the `Function` query-node's *name* property to "Foo*", the *name* properties of the function's `Parameter` children-queries to "Bar", and group the latter queries in an *OR* query. A global `query()` function applies a given query-tree to a fact database, yielding a selection of the matched ASG nodes.

Our design has several advantages. First, we can build simple queries *e.g.* "show all variables called `x`" or complex ones *e.g.* "show all templates whose second type-parameter virtually inherits class `T`" just by assembling atomic query-nodes. Second, we store query-trees in XML, and provide a query editor, so users can edit queries on-the-fly, without recompilation. Third, we can implement many code metrics, or code smells, simply by counting the results of their related queries. For instance, a simple code complexity metric returns the number of queried decision-points (*e.g.* `if`, `for`, `return`, ... statements) in a piece of code. A careful implementation of the `query()` interface, using early termination and fast node retrieval by node-by-type-hashing, allows querying millions of ASG nodes in subsecond time.

2.3. Visualization Tools

Figure 2 shows several views of our IRE. The *project view* allows setting up an extraction project, just as a classical build project in *e.g.* Visual C++. The *fact database* shows the fact files created by the *SolidFX* parser. The *code view* shows the source code, reconstructed from the ASG. Users can point-and-click this code to select queries (or metrics) from the available query (or metric) library matching the clicked node's type, and apply these. Query results are shown in a separate *selection view*, similar to a clipboard. When the user activates (picks) a selection here, its ASG nodes (and their metrics, if any), are shown in a table-like *metric view*, and also in the code view, by coloring the nodes' source code, as shown in the figure. We also provide *graph views* that render selections as graphs using various layouts. The UML view in Fig. 2 shows a class inheritance graph. The graph is extracted using a closure-query over a function-call query, and is laid out using the GraphViz library [1]. All views read selections, and the fact extractor and query-system create selections, so we can easily and orthogonally combine analyses, queries, and visualizations on-the-fly, using only the environment's GUI.

Scale is a major problem when reverse engineering real-world systems. As explained, the SOLIDFX parser and query system can easily handle million-LOC C/C++ code bases. We add scalability to our visualizations by several techniques. First, we add a zoom-slider to the code view. When zooming out, each code line is rendered as a pixel line, colored by the desired metric and/or selection attributes (Fig. 2, bottom). As explained in [3], this technique can show overviews of thousands of lines on a single screen. We also added a similar feature, called a *table lens*, to the tables showing code metrics. When zooming out, each table cell becomes a pixel bar colored and/or scaled to reflect its metric value. This reduces tables of thousands of rows to compact, easy-to-follow graphs [6]. Sorting the columns, we can easily detect outliers of interest *e.g.* the most complex, buggy, or smelly code. Since all views are linked, clicking on any element in a view highlights it in all other views, thereby making complex assessments easy.

2.4. Tool Integration

We support two types of tool integration. First, we provide an open ASG and query API (written in C++) to our proprietary fact database format, so developers can write plug-ins for our environment. This is how we integrated the table-lens tool. Using a proprietary database format allows us to load/save facts much faster (and more compactly) than using a generic database, *e.g.* MySQL. Second, we provide several exporters for the fact database to formats such as XML (for the ASG), XMI (for UML diagrams), SQL (for metrics), and Dot and VCG (for general graphs). For example, the inset in Fig. 2 shows a C++ system call graph, drawn by the hierarchical edge-bundling tool of Holten [4], connected via a data exporter. This view quickly shows the strong dependency between two halves of the considered system, visible as the thick edge bundle in the middle.

2.5. User Experience

We have used our IRE for reverse-engineering open-source (Linux kernel, wxWidgets, Quake, Python, Boost, STL) and also commercial code (undisclosed). For the same tasks, we previously used the same view and parser tools, but not integrated as an environment. For the *parsing* phase, the environment was not much more effective. Here, a simple text

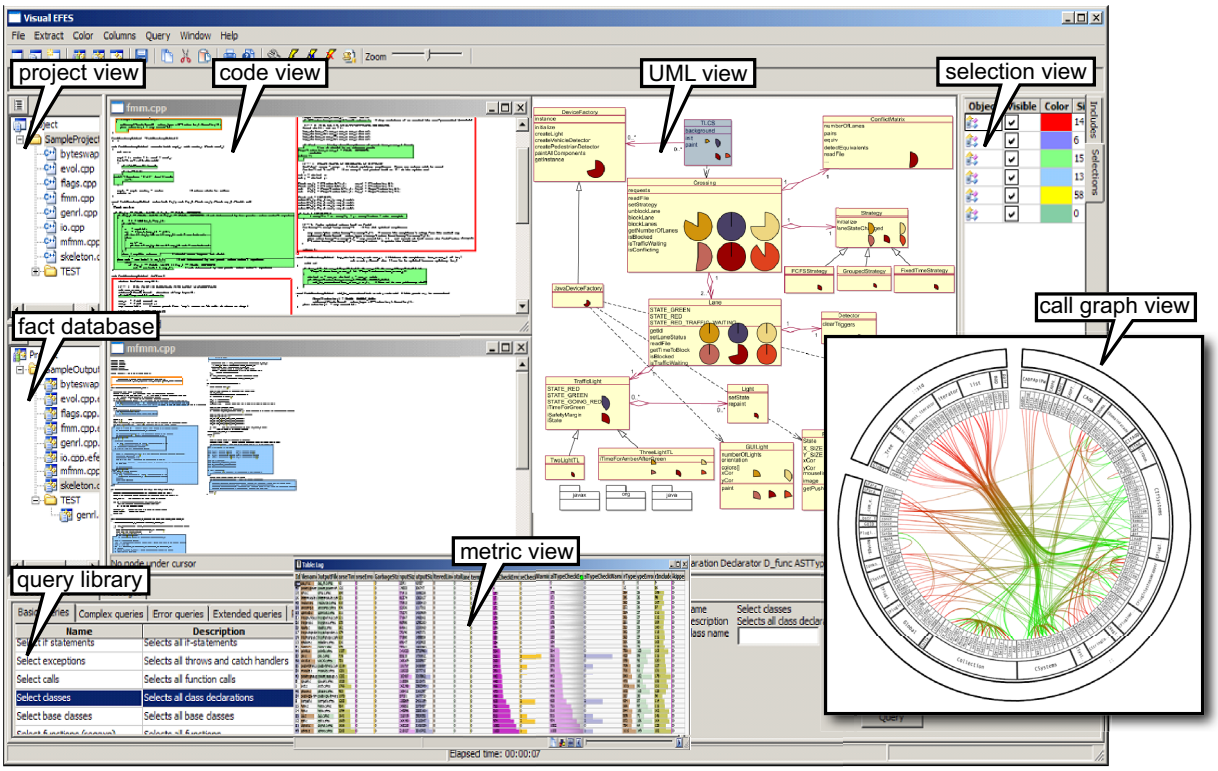

Figure 2. Integrated reverse-engineering environment views

makefile-like project was sufficient. However, for the *exploration* phase, the IRE and its tight tool integration were massively more productive than using the same tools standalone, connected by little scripts and data files. Reverse-engineering and maintainability and quality assessment scenarios are by nature iterative and exploratory, so they map perfectly to repeated selection, query, and interactive visualization operations.

Our users often asked if we could provide the functionality shown here with Eclipse and/or existing C++ extractors, besides ours. Integrating our SOLIDFX extractor in Eclipse should be relatively simple, but integrating all views shown here will be very complex, as they use advanced graphics features unsupported in Eclipse. Using other C++ extractors is also complex, as few provide the fine-grained, efficient, comprehensive fact-querying API needed to give the support level described here. Actually, the only environment we know which could provide similar functions is Visual Studio 2005, which recently started opening its compiler's low-level APIs for tool builders.

3. Conclusions

We have presented an integrated reverse-engineering environment (IRE) built by combining a tolerant C/C++ parser, an open query API, and several visualization tools. Our environment promotes the same way of working in reverse engineering as in classical forward-engineering IDEs, thereby making reverse engineering activities fast and easy to learn and perform by end users. We are currently use our environment in several large reengineering projects, in order to assess and improve its effectiveness and efficiency.

References

[1] AT&T. The GraphViz package. `www.graphviz.org`.

[2] F. Boerboom and A. Janssen. Fact extraction, querying, and visualization of large C++ code bases. In *Master's Thesis, Dept. of Computer Science, Eindhoven University, Netherlands*, 2006, 105 pages.

[3] S. Eick, S. Steffen, and E. Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Soft. Eng.*, 18(11):957–968, 1992.

[4] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Trans. Visualization and Computer Graphics*, 12(5):741–748, 2006.

[5] S. McPeak. The Elsa C++ parser. 2006. `www.cs.berkeley.edu/~smcpeak`.

[6] A. Telea. Combining extended table lens and treemap techniques for visualizing tabular data. In *Proc. EuroVis*, pages 51–58. IEEE, 2006.