A Visual Analytics Toolset for Program Structure, Metrics, and Evolution Comprehension

Dennie Reniers[a], Lucian Voinea[a], Ozan Ersoy[b], Alexandru Telea[b,*]

[a]*SolidSource BV, Eindhoven, the Netherlands*
[b]*Institute Johann Bernoulli, University of Groningen, the Netherlands*

Abstract

Software visual analytics (SVA) tools combine static program analysis and repository mining with information visualization to support program comprehension. However, building efficient and effective SVA tools is highly challenging, as it involves software development in program analysis, graphics, information visualization, and interaction. We present a SVA toolset for software maintenance, and detail two of its components which target software structure, metrics and code duplication. We illustrate the toolkit's application with several use cases, discuss the design evolution of our toolset from a set of research prototypes to an integrated, scalable, and easy-to-use product, and argue how these can serve as guidelines for the development of future software visual analytics solutions.

Keywords: Software visualization, static analysis, visual tool design

1. Introduction

Software maintenance covers 80% of the cost of modern software systems, of which over 40% represent software understanding [1, 2]. Although many visual tools for software understanding exist, most know very limited acceptance in the IT industry. Key reasons for this are limited scalability of visualizations and/or size of datasets, long learning curves, and poor integration with established software analysis or development toolchains [3, 4, 5].

Visual analytics (VA) integrates graphics, visualization, interaction, data analysis, and data mining to support reasoning and sensemaking for complex problem solving in engineering, finances, security, and geosciences [6, 7]. These fields share many similarities with software maintenance in terms of *data* (large databases, highly structured text, and graphs), *tasks* (sensemaking by hypothesis creation, refinement, and validation), and *tools* (combined analysis and visualization). VA explicitly addresses tool scalability and integration, as opposed to pure data mining (whose main focus is scalability) or information visualization (InfoVis, mainly focusing on presentation). As such, VA is a promising model for building effective and efficient software visual analysis (SVA) tools. However, building VA solutions for software comprehension is particularly challenging, as developers have to master technologies as varied as static analysis, data mining, graphics, information visualization, and user interaction design.

In this paper, we present our experience in building SVA tools for software maintenance. We outline the evolution path from a set of research prototypes to a commercial toolset which is used by many end-users in the IT industry. Our toolset supports static analysis, quality metrics computation, clone detection, and research-grade InfoVis techniques such as table lenses, bundled graph layouts, cushion treemaps, and dense pixel charts. The toolset contains several end-user applications, of which we focus here on two: visual analysis of program structure and code duplication. These applications share elements at both design and implementation level, and can be used separately or combined to support tasks such as detecting correlations of structure, dependencies, and quality metrics; assessing system modularity; and planning refactoring.

*Corresponding author

Email addresses: dennie.reniers@solidsource.nl (Dennie Reniers), lucian.voinea@solidsource.nl (Lucian Voinea), o.ersoy@rug.nl (Ozan Ersoy), a.c.telea@rug.nl (Alexandru Telea)

URL: www.solidsource.nl (Dennie Reniers), www.solidsource.nl (Lucian Voinea), www.cs.rug.nl/~alext (Alexandru Telea)

The contributions of this paper are as follows:

- present the design decisions and evolution path of a SVA toolset for program comprehension from research prototypes into an actual product;
- present the lessons learned in developing our toolset in both research and industrial contexts, with a focus on efficiency, effectiveness, and acceptance;
- present supporting evidence for our design decisions based on actual usage in practice.

Section 2 introduces software visual analytics. Section 3 details the general architecture of our toolset. Section 4 details the toolset's components for data mining and visualization of software structure, metrics, and code duplicates. Section 5 illustrates the usage of our toolset in a real-world industrial software assessment case. Section 6 discusses the challenges of developing efficient, effective, and accepted SVA tools from our toolset's experience. Finally, section 7 concludes the paper.

2. Related Work

Software visual analytics can be roughly divided into data mining and visualization, as follows.

Data mining covers the extraction of raw data from source code, binaries, and source control management (SCM) systems such as CVS, Subversion, Git, CM/Synergy, or ClearCase. Raw data delivered by static syntax and semantic analysis is refined into structures such as call graphs, control flow graphs, program slices, code duplicates (clones), software quality metrics such as complexity, cohesion, and coupling, and change patterns. Static analyzers can be divided into *lightweight* ones, which use a subset of the target language grammar and semantics and typically trade fact completeness and accuracy for speed and memory; and *heavyweight* ones, which do full syntactic and semantic analysis at higher cost. Well-known static analyzers include LLVM, Cppx, Columbus, Eclipse CDT, and Elsa (for C/C++), Recoder (for Java), and ASF+SDF (a meta-framework usable with different language-specific front-ends) [8]. Metric tools include CodeCrawler [9], Understand, and Visual Studio Test Suite (VSTS). An overview of static analysis tools with a focus on C/C++ is given in [10]. Practical insights in software evolution and software quality metrics are given in [11, 12, 13].

Software visualization (SoftVis) uses information visualization (InfoVis) techniques to create interactive displays of software structure, behavior, and evolution. Recent trends in SoftVis include scalable InfoVis techniques such as treemaps, icicle plots, bundled graph layouts, table lenses, parallel coordinates, multidimensional scaling, and dense pixel charts to increase the amount of data shown to the user at a single time. An excellent overview of software visualization is given in [14]. Well-known software visualization systems include Rigi, VCG, aiSee, and sv3D (for structure and metrics); and CodeCity and SeeSoft (for software evolution).

Although tens of new software analysis and visualization tools emerge every year from research, as shown by the proceedings of *e.g.* ACM SOFTVIS, IEEE VISSOFT, MSR, ICPC, WCRE, and CSMR, building *useful* and *used* tools is difficult. Reasons cited for this include the small return-on-investment and recognition of tools in academia (as opposed to papers), high maintenance cost, and high ratio of infrastructure-to-novelty (truly usable tools need fine-tuned implementations, help modules, and platform-independence, while research prototypes can focus on novelty) [5, 15]. Combining analysis and visualization in the same tool makes development only more complex, so good design patterns and guidelines are essential.

Given all these, how to bridge the gap between SVA tool prototypes and actual efficient and effective products? And how to combine analysis and visualization software in maintainable tool designs?

3. Toolset Architecture

In the past decade, we have built over 20 SVA tools for software requirements, architecture, behavior, source code, structure, dependencies, and evolution. We used these tools in academic classes, research, and industry, in groups from a few to tens of users. Latest versions of our tools have formed the basis of SolidSource, a company specialized in software visual analytics [16]. Table 1 outlines our most important tools (for a full list and the actual tools, see [17]). We next present the latest version of our toolset, discuss the design decisions and lessons learned during the several years of its evolution, and illustrate its working with two of its most recent tools.

Tool	Targeted data types	Visual techniques	Analysis techniques
SoftVision (2002) [18]	software architecture	node-link layouts (2D and 3D)	none (visualization tool only)
CSV (2004) [19]	source code syntax and metrics	pixel text, cushions	C++ static analysis (`gcc` based parser)
CVSscan [20] (2005)	file-level evolution	dense pixel charts annotated text	CVS data mining (authors and line-level changes)
CVSgrab (2006) [21]	project-level evolution	dense pixel charts, cushions	CVS/SVN data mining (project-level changes)
MetricView (2006) [22]	UML diagrams and quality metrics	node-link layouts (2D), table lenses	C++ lightweight static analysis (class diagram extraction)
MemoView (2007) [23]	software dynamic logs (memory allocations)	table lenses, timelines, cushions	C runtime instrumentation (`libc` malloc/free interception)
SolidBA (2007) [24]	build dependencies and build cost metrics	table lenses, node-link layouts (2D)	C/C++ build dependency mining and automated refactoring
SolidFX (2008) [25]	reverse engineering	pixel text, table lenses, annotated text	C/C++ heavyweight error-tolerant static static analysis
SolidSTA (2008) [16]	file and project-level evolution	dense pixel charts, cushions, timelines	CVS/SVN/Git data mining and source code metrics
SolidSX (2009) [16]	structure, associations, metrics	HEB layouts, treemaps, table lenses, cushions	.NET, C++, Java lightweight static analysis
SolidSDD (2010) [16]	code duplicates, code structure, metrics	HEB layouts, treemaps, table lenses, pixel text	C, C++, Java, C# parameterizable syntax-aware clone detection

Table 1: Software visual analytics tools - evolution history. Tools discussed in this paper are in bold (Sec. 4)

Our toolset uses a simple dataflow architecture (Fig. 1). Raw input data comes in two forms: non-versioned source code or binaries, and versioned files stored software repositories. From raw data, we extract several *facts*: syntactic and semantic structure, static dependency graphs, and source code duplication. Relational data is stored into a SQLite database whose entries point to flat text files (for actual source code content) and binary files (for complete syntax trees, see 4.1.1). Fact extraction is implemented by specific tools: parsers and semantic analyzers for source code, binary analyzers for binary code, and clone detectors for code duplication (see Sec. 4).

Besides facts, our database stores two other key elements: selections and metrics. *Selections* are sets of facts (or other selections) and support the iterative data refinement in the so-called visual sensemaking cycle of VA [6, 7]. They are created either by tools, *e.g.* extraction of class hierarchies or call graphs from annotated syntax graphs (ASGs), or by users during interactive data analysis. Selections have unique names by which they are referred by their clients (tools) and also persistently saved. *Metrics* are numerical, ordinal, or categorical attributes which can be associated to facts or selections, and are computed either by tools (*e.g.* complexity, fan-in, fan-out, cohesion, and coupling) or added as annotations by users. Each selection is stored as a separate SQL table, whose rows are its facts, and columns are the fact attributes (unique ID, type, location in code or binary, and metrics). This model is simple and scales well to fact databases of hundreds of thousands of facts with tens of metrics. Trees and graphs can also be stored as SQL tables in terms of adjacency matrices. Simple queries and filters can directly be implemented in SQL. More complex queries have the same interface as the other components: they read one or more selections and create an output selection.

Selections are the glue that allows composing multiple analysis and visualization tools. All analysis and visualization components in our toolkit are weakly typed: They all read selections, and optionally create selections (see Fig. 1). In this way, existing or new tools can be flexibly composed, either statically or at run-time, with virtually no configuration costs. To ensure consistency, each component decides internally whether (and how) it can execute its function on a given input selection.

Visualizations display selections and allow users to interactively navigate, pick elements, and customize the visual aspect. Since they only receive selections as inputs, they 'pull' their required data on demand as needed, *e.g.* a source code viewer opens the files referred by its input selection to get the text to render. Hence, data can be efficiently handled by reference (selections referred to by name), but tools can also decide to cache data internally to minimize traffic with the fact database, if so desired. However, this decision is completely transparent at the toolkit level.

After lengthy experimentation with many types of visualizations, we limited ourselves to a small subset hereof, as follows. *Table lenses* show large tables by zooming out the table layout and displaying cells as pixel bars scaled and colored by data

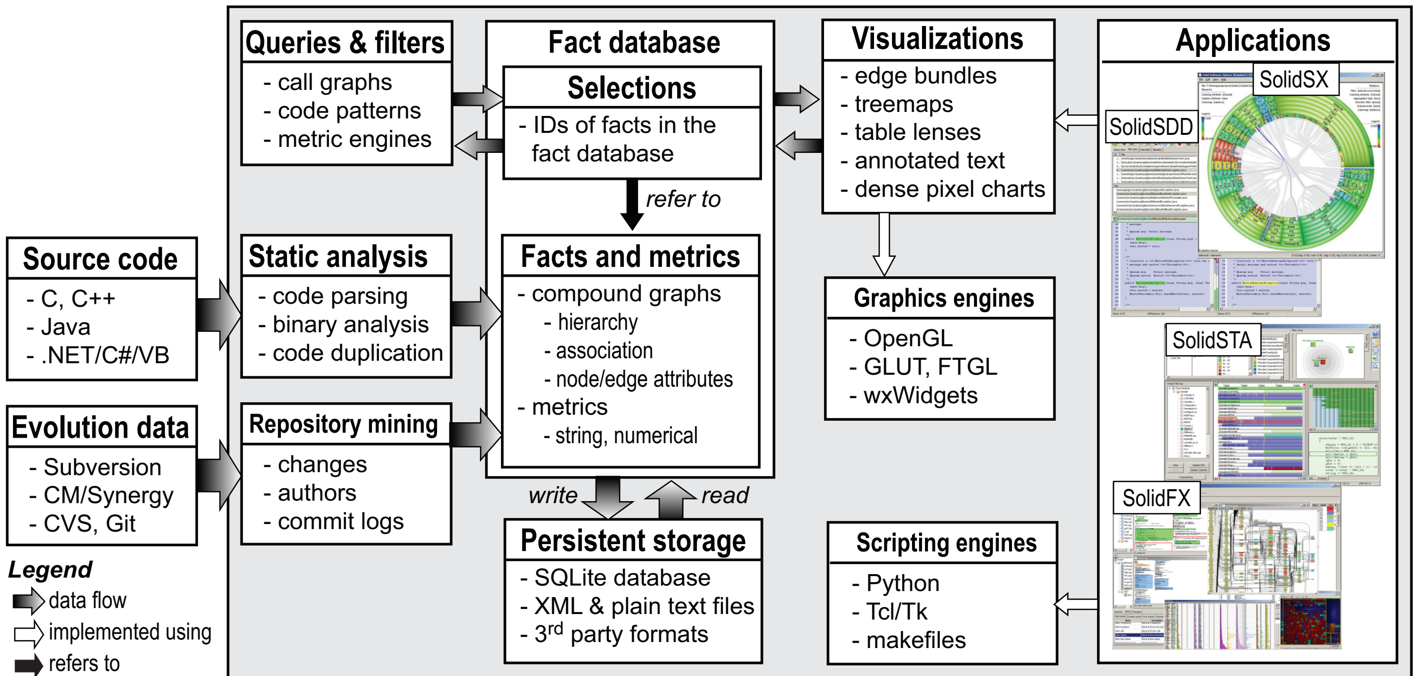


Figure 1: Toolset architecture (see Section 3)

values [26]. Subpixel sampling techniques enhance this idea allowing to visualize tables up to hundreds of thousands of rows on a single screen [27]. *Hierarchically bundled edges* (HEBs) compactly show compound (structure and association) software graphs, *e.g.* containinment and call relations, by bundling association edges with structure data [28]. *Squarified cushion treemaps* show software structure and metrics scalably for up to tens of thousands of elements on a single screen [29].

A single treemap, HEB, or table lens can show the correlation of just a few metrics and/or relations. To augment this, we use the well-known *multiple correlated views* InfoVis concept. Each view can have its own user-specified input selection and visual look-and-feel. Mouse-based picking is synchronized between all views: selecting items in one view updates a 'user selection' which is monitored by other views, using an Observer pattern. Besides the above, our toolset also provides classical visualizations: metric-annotated source code text, tree browsers, customizable color maps, legends, annotations, timelines, details-on-demand at the mouse cursor, and text and metric-based search facilities.

4. Toolset Components

Our toolset consists of several end-user applications which share analysis and visualization components. We next describe two of these applications: the SolidSX structure analyzer (Sec. 4.1) and the SolidSDD duplication detector (Sec. 4.1). The fully operational tools, including Windows-based installers, manuals, videos, and sample datasets are freely available for researchers from [16].

4.1. SolidSX: Structural Analysis

The SolidSX (Software eXplorer) supports the task of analyzing software structure, dependencies, and metrics. Several built-in parsers are provided: Recoder for Java source and bytecode files [30], Reflector for .NET/C# assemblies [31], and Microsoft's free parser for Visual C++ *.bsc* symbol files. Built-in filters refine parsed data into a compound attributed graph consisting of one or several hierarchies, *e/g.* folder-file-class-method containment and namespace-class-method containment; dependencies, *e.g.* calls, symbol usage, inheritance, and package or header inclusion; and basic metrics, *e.g.* code and comment size, cyclomatic complexity, fan-in, fan-out, and symbol source code location.

4.1.1. Static Analysis: Ease of Use

Setting up static code analysis is notoriously complex, error-prone, and time consuming. For .NET/VB/C#, Java, and Visual C++, we succeeded in *completely* automating this process. The user is only required to input a root directory for code and, for Java, optional class paths. C/C++ static analysis beyond Visual Studio proves highly challenging. SolidSX can read static information created by our separate SolidFX C/C++ static analyzer [25]. Although SolidFX is scalable to millions of lines of code, covers several dialects beyond Visual C++ (gcc, C89/99, ANSI C++), robustly handles incorrect and incomplete code, integrates a preprocessor, provides a Visual C++ project file parser, and uses so-called compiler wrapping to emulate the `gcc` and Visual C++ compilers [32], users still typically need to manually supply many per-project defines and include paths. Moreover, compiler wrapping requires the availability of a working build system on the target machine. The same is true for other heavyweight parsers such as Columbus [32] or Clang [33].

A second design choice regards performance. For Java, Recoder is close to ideal, as it delivers heavyweight information with 100 KLOC/second parse speed on a typical PC computer. Strictly speaking, visual structure-and-metric analysis only needs lightweight analysis (which is fast) as the information graininess typically does not go below function level. For .NET, the Reflector lightweight analyzer is fast, robust, and simple to use. The same holds for Microsoft's *.bsc* symbol file parser.

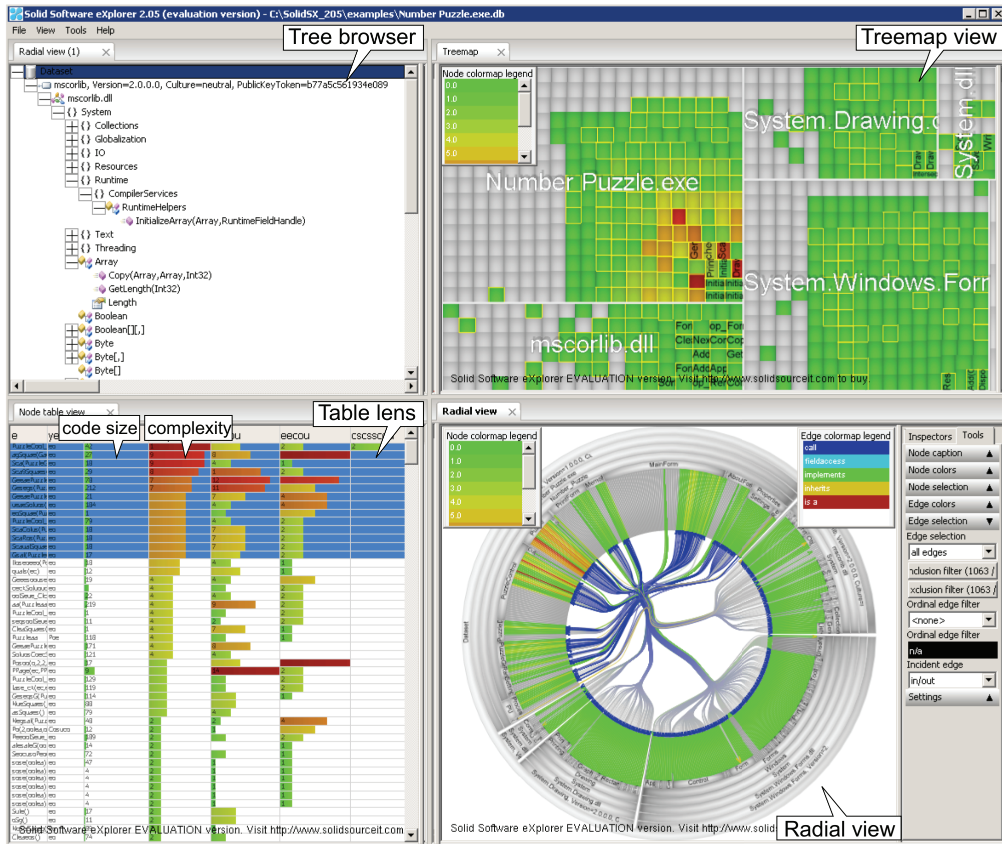
For C/C++ beyond Visual Studio, a lightweight, robust, easy-to-use analyzer is still not available. After experimenting with many such tools, *e.g.* CPPX [34], gccxml, and MC++, we found that these often deliver massively incorrect information, mainly due to simplified preprocessing and name lookup implementations. The built-in C/C++ parsers of Eclipse CDT, KDevelop, QtCreator, and Cscope [35] are somehow better in correctness, and can also work incrementally upon code changes. However, these parsers depend in complex ways on their host IDEs and do not have well-documented APIs. Extended discussions with Roberto Raggi, the creator of the KDevelop and QtCreator parsers, confirmed this difficulty.

4.1.2. Structure Visualization

SolidSX offers four views (Fig. 2 top): classical tree browsers, table lenses of node and edge metrics, treemaps, and a HEB layout for compound graphs [28]. All visualizations have carefully designed *presets* which allow one to use them with no additional customization. They all depict selections from the fact database created by the code analysis step. Users can also create selections in any view by either direct interaction or custom queries. These two mechanisms realize the linked view concept, which enables users to easily create complex analyses of correlations of structure, dependencies, and metrics along different viewpoints. Figure 2 top) illustrates this on a C# system of around 45000 LOC (provided with the tool distribution). The HEB view shows function calls over system structure: caller edge ends are blue, callee edge ends are gray. Node colors show McCabe's code complexity metric on a green-to-red colormap, thereby enabling complexity correlation with the system structure. We see that the most complex functions (warm colors) are in the module and classes located top-left in the HEB layout. The table lens view shows several function-level code metrics, and is sorted on decreasing complexity. This allows one to see how different metrics correlate with each other. Alternatively, one can select *e.g.* the most complex or largest functions and see them highlighted in the other views. The treemap view shows a flattened system hierarchy (modules and functions only), with functions ordered top-down and left-to-right in their parent modules on code size, and colored on complexity. The visible 'hot spot' indicates that complexity correlates well with size. Constructing the *entire* scenario, including the static analysis, takes about 2 minutes and under 20 mouse clicks.

4.1.3. Toolchain Integration

Similarly to Koschke [5], we discovered that *integration* in accepted toolchains is a key acceptance factor. To ease this process, we added a *listener* mechanism to SolidSX. The tool listens for command events sent asynchronously on a specified TCP/IP port, *e.g.* load a dataset, zoom on some data subset, change view parameters, and also sends user interaction command events to a second port (*e.g.* user has selected some data). This allows integrating SolidSX in any third-party tool or toolchain by building thin wrappers which read, write, and process desired events. No access to the tool's source code is required. For example, we integrated SolidSX in the Visual Studio IDE by writing a thin plug-in of around 200 LOC which translates between the IDE and SolidSX events (Fig. 2 bottom). Selecting and browsing code in the two tools is now in sync. The open SQLite database format further simplifies integration at data level. Integrating SolidSX in Eclipse, KDevelop, and QtCreator is under way and is relatively easy, once we finalize the data importers from these IDEs' proprietary fact databases into our SQL database.



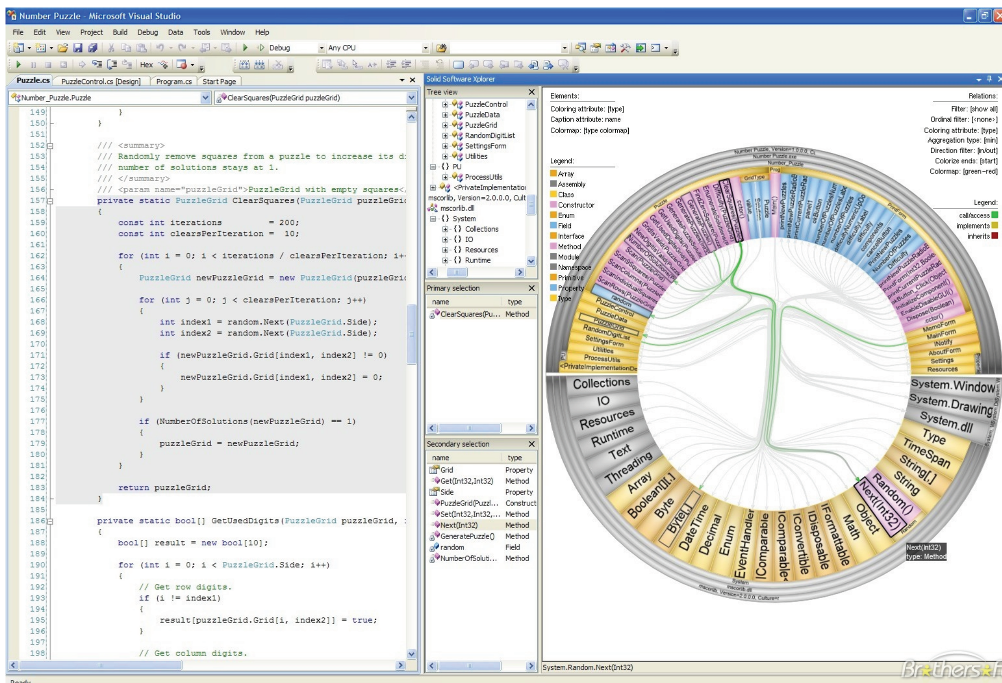


Figure 2: Top: SolidSX views (tree browser, treemap, table lens, radial HEB); Bottom: Visual Studio integration of SolidSX using a socket-baset mechanism (see Section 4.1.3)

4.2. *SolidSDD: Clone Detection*

Code duplication detection, or clone detection, is an important step in maintenance tasks such as refactoring, redocumentation, architecture extraction, and test case management. Although hundreds of papers on this topic exist, only few clone detection *tools* offer the clone information in effective ways for assisting refactoring activities.

We addressed the above by developing SolidSDD (Software Duplication Detector). SolidSDD implements an extended version of the CCfinder tool [36] which combines lightweight syntax analysis and token-based detection. The detection is user-configurable by clone length (in statements), identifier renaming (allowed or not), size of gaps (inserted or deleted code fragments in a clone), and whitespace and comment filtering. However, the main novelty in SolidSDD is in how clones are visualized.

Given a source code tree (C, C++, Java, or C#), SolidSDD constructs a SQL database containing a duplication graph, in which nodes are cloned code fragments (in the same or different files) and edges indicate clone relations. Hierarchy information is added to this graph either automatically (from the code directory structure) or from a user-supplied dataset (*e.g.* from static analysis). The result is a compound graph. Nodes and edges can have metrics, *e.g.* percentage of cloned code, number of distinct clones, and whether a clone includes identifier renaming or not. Metrics are automatically aggregated bottom-up using the hierarchy information.

Figure 3 shows SolidSDD in use to find code clones in the well-known Visualization Toolkit (VTK) library [37]. On VTK version 5.4 (2420 C++ files, 668 C files, and 2660 headers, 2.1 MLOC in total), SolidSDD found 946 clones in 280 seconds on a 3 GHz PC with 4 GB RAM. For replication ease, we simply used the default clone detection settings of the tool. Figure 3 a shows the code clones atop of the system structure with SolidSX. Hierarchy shows the VTK directory structure, with files as leaves. Edges show aggregated clone relations between files: file a is connected to file b when a and b share at least one clone. Node colors show the total percentage of cloned code in the respective subsystem with a green-to-red (0..100%) colormap. Edge colors show percentage of cloned code in the clone relations aggregated in an edge. Figure 3 a already shows that VTK has quite many intra-system clones (short edges that start and end atop of the same rectangle) but also several inter-system clones (longer edges that connect rectangles located in different parts of the radial plot). Inter-system clones are arguably less desirable.

Three subsystems have high clone percentages: *examples* (S_1), *bin* (S_2) and *Filtering* (S_3). Browsing the actual files, we saw that almost all clones in *examples* and *bin* are in sample code files and test drivers which contain large amounts of copy-and-paste. Clones in *Filtering*, a core subsystem of VTK, are arguably more worth removing. In Fig. 3 b, we use SolidSX's zoom and filter functions to focus on this subsystem and select one of its high-clone-percentage files f (marked in black) which has over 50% cloned code. When we select f, only its clone relations are shown. We call the set of files f_c with which f shares clones the *clone partners* of f. We see that all clone partners of f are located in the same *Filtering* subsystem, except one (g) which is in the *Rendering* subsystem.

Figure 3 c shows additional functions offered by SolidSDD. The top light-blue table shows all system files with several metrics: percentage of cloned code, number of clones, and presence of identifier renaming in clones. SolidSDD's views are linked with SolidSX's views via the socket mechanism outlined in Sec. 4.1.3, so the selected file f in Fig. 3 c is also highlighted in this table in red. The table below shows the clone partner files f_c of f. In this table, we can identify the file g which shares clones with f but is in another subsystem. We also see here that g contains about 50% of code cloned from f. We select g and use the two text views (at the bottom of Fig. 3 b) to examine in detail all clones between f and g. The left view shows the first selected file (f) and the right view the selected clone partner (g). Scrolling of these views is automatically synchronized so one can easily follow corresponding code fragments. Text is color-coded as follows: non-cloned code (white), code from f which is cloned in g (light blue), renamed identifier pairs (green in left view, yellow in right view), and code from f which is cloned but whose clones are located in some other file than g (light brown). The last color allows us to navigate from f to other clone partners: Clicking on the light brown code in the left text view (f) in Fig. 3 c replaces the file g in the right view by that clone partner of f (let us call it h) with which f shares the brown code, and also selects h in the clone partner list view. Fig. 3 d shows this perspective. We now notice that the code in f which is part of the clone $f - g$ (light blue in Fig. 3 c) is *included* in the clone $f - h$ (light blue Fig. 3 c).

The SolidSDD-SolidSX combination offers many other perspectives and analyses, as detailed by its manual. Fig. 3 e shows a final example. We use SolidSX's table lens to sort all files by percentage of cloned code and zoom out to see a distribution of this metric (sixth column from right). We see that around 30% of all files contain some amount of cloning. Interactively selecting the top 20% files in the table lens highlights all files having >80% cloned code in the radial system view in black - we

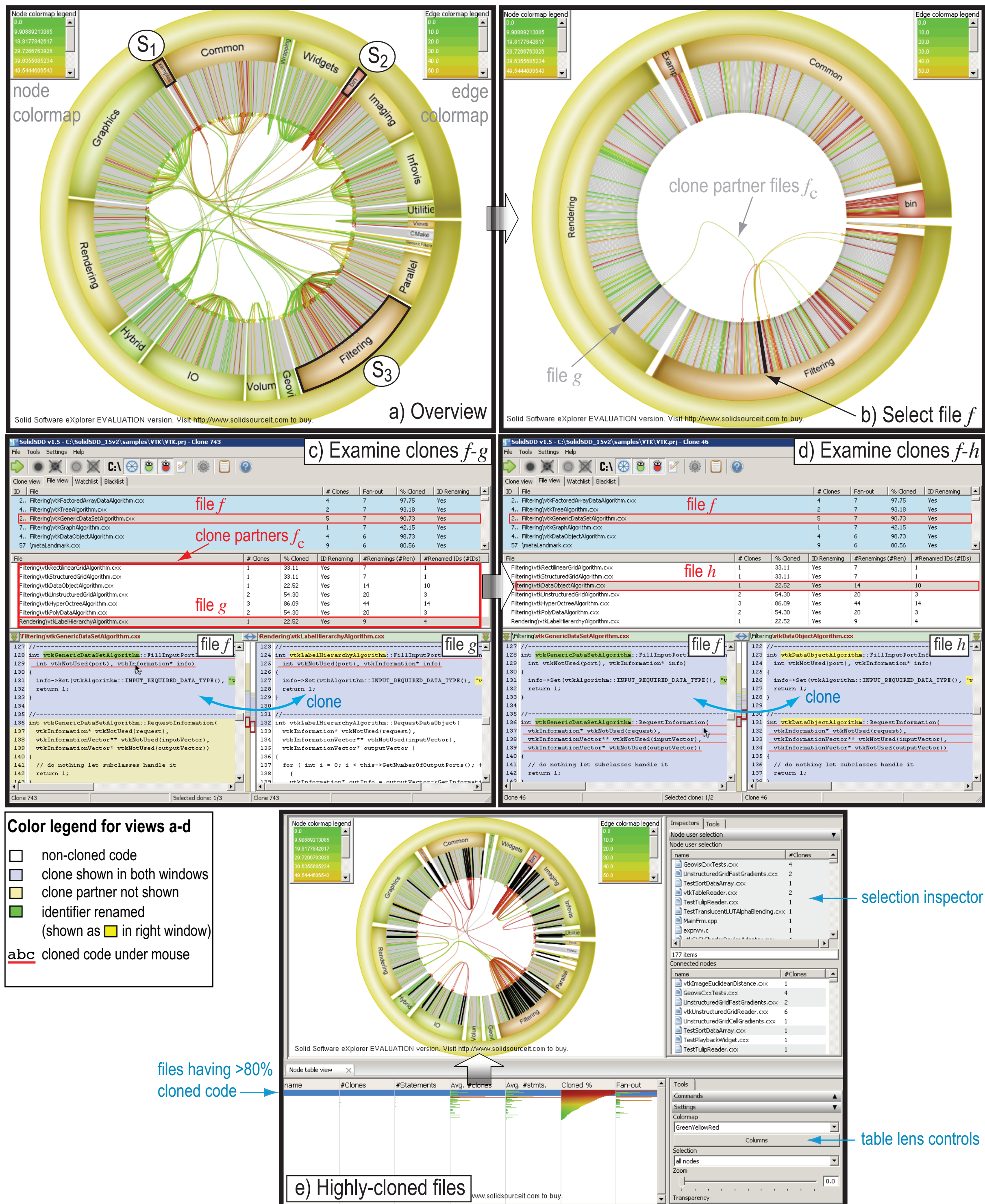


Figure 3: SolidSDD clone detector visual analysis of the VTK code base (see Sec. 4.2)

see that every single subsystem in VTK contains such files. Details on the selected files are shown in the selection inspector at the right. This type of visualization supports refactoring planning by giving insight on where recoding activities would need to take place in order to *e.g.* remove the largest clones.

5. Toolset Assessment in Practice

We have used our SVA toolset in many applications, both in research [19, 38] and the industry [24, 39, 25]. We next briefly describe one application which outlines the added value of strong interplay between the two tools discussed so far. To further outline the extensibility of our toolset, this application briefly introduces one additional tool: SolidSTA, a visual analytics application for software evolution data mining. SolidSTA uses the same dataflow architecture (Sec. 3) and followed the same development path from a research prototype [21] to a scalable, easy-to-use tool as SolidSX and SolidSDD. A detailed description of SolidSTA is provided in [21]. The tool is freely available for researchers from [16].

A major automotive company developed an embedded software stack of 3.5 million lines of C code in 15 releases over 6 years with three developer teams in Western Europe, Eastern Europe, and Asia. Towards the end, it was seen that the project could not be finished on schedule and that new features were hard to introduce. The management was not sure what went wrong. The main questions were: was the failure caused by bad architecture, coding, or management; and how to follow up - start from scratch or redesign the existing code. An external consultant team performed a post-mortem analysis using our toolset (SolidSDD, SolidSTA, SolidSX). The team had only *one week* to deliver its findings and only the code repository as information source. For full details, see [40].

The approach involved the classical VA steps: data acquisition, hypothesis creation, refinement, (in)validation, and result aggregation and presentation (Fig. 4). First, we mined change requests (CRs), commit authors, static quality metrics, and call and dependency graphs from the CM/Synergy repository into our toolset's SQL fact database using SolidSTA (1). Next, we examined the distribution of CRs over project structure. Several folders with many open CRs emerged (red treemap cells in Fig. 4 (2)). These correlate quite well with the team structure: the 'red' team owns most CRs (3). To further see if this is a problem, we looked at the CR distribution over files over time. In Fig. 4 (4), files are shown as gray lines vertically stacked on age (oldest at bottom), and CRs are red dots (the same layout is used *e.g.* in [21]). The gray area's shape shows almost no project size increase in the second project half, but many red dots over *all* files in this phase. These are CRs involving old files that were never closed. When seeing these images, the managers instantly recalled that the 'red' team (located in Asia) had lasting communication problems with the European teams, and acknowledged that it was a mistake to assign so many CRs to this team.

We next analyzed the evolution of various quality metrics: fan-in, fan-out, number of functions and function calls, and average and total McCabe complexity. Static analysis is done using our heavyweight analyzer SolidFX [25]. Since this is a one-time analysis rather than an incremental change-and-reanalyze path, speed and configuration cost are not essential, so most of the analyzers listed in Sec. 4.1.1 could be used equally well. The graphs in (5) show that these metrics have a slow or no increase in the second project half. Hence, the missed deadlines were not caused by code size or complexity explosion. Yet, the average complexity per function is high, which implies difficult testing. This was further confirmed by the project leader.

Finally, to identify possible refactoring problems, we analyzed the project structure using SolidSX. Fig. 4 (6) shows disallowed dependencies, *i.e.* modules that interact bypassing interfaces. Fig. 4 (7) shows modules related by mutual calls, which violate the product's desired strict architectural layering. These two views suggest difficult step-by-step refactoring and also difficult unit testing. Again, these findings were confirmed by the project leaders.

6. Discussion

Based on our SVA tool building experience, we next try to answer several questions of interest for our audience[1]. To clarify the standpoints taken, we first introduce the concept of a tool *value model*, along the lines of the well-known lean development cost model [41]: We state that a SVA tool is useful if it delivers high *value* with minimal *waste* to its stakeholder user. Different users (or stakeholders) will thus assess the same tool differently as they have different value and waste models [42]. Hence, rhe answers to our questions of interest are strongly dependent on the stakeholder perspective chosen, as follows.

[1]as taken from the WASDeTT 2010 call for papers (www.info.fundp.ac.be/wasdett2010)

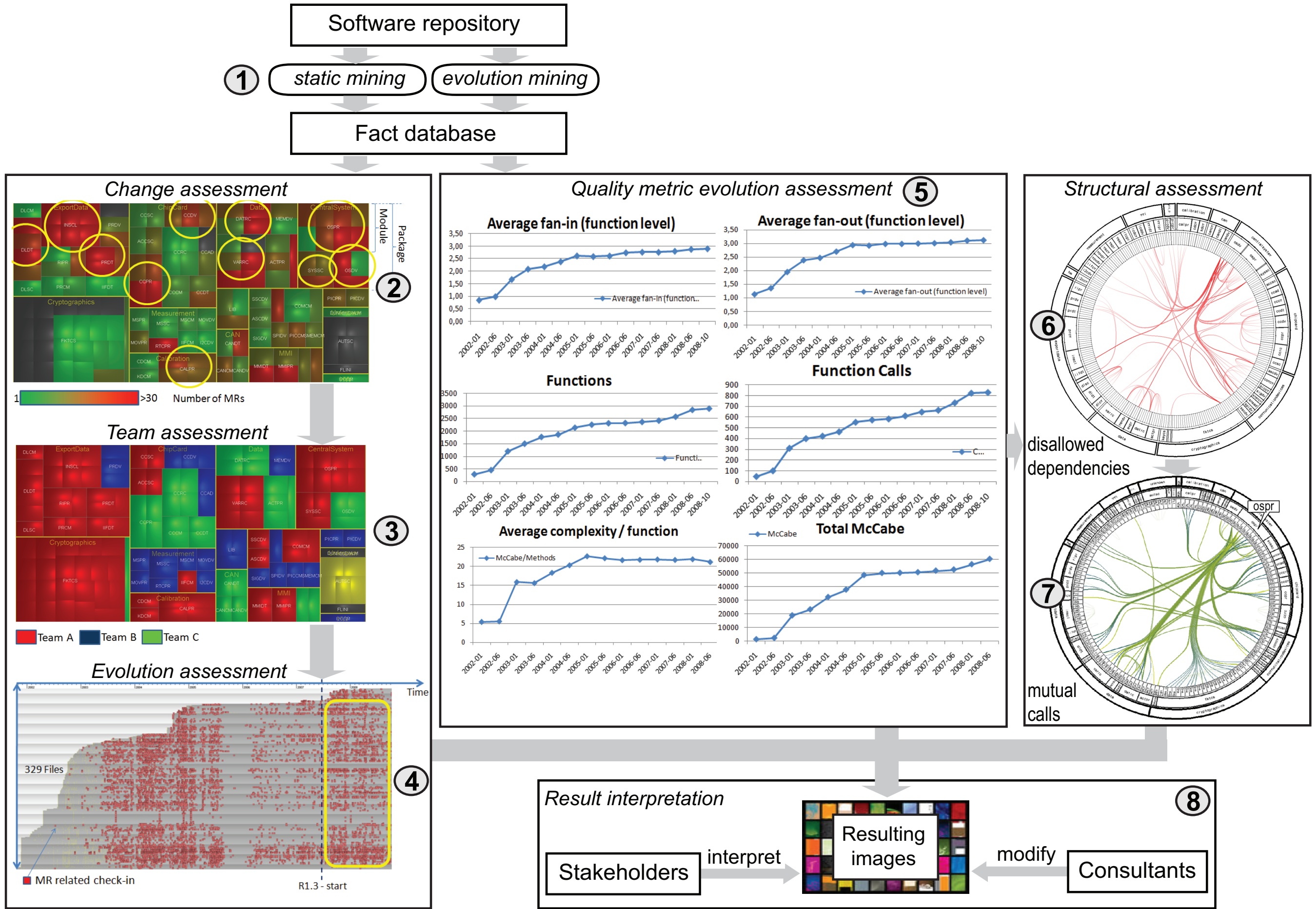


Figure 4: Data collection, hypothesis forming, and result interpretation for product and process assessment (Sec. 5). Arrows and numbers indicate the order of the performed steps

6.1. *Should academic tools be of commercial quality?*

We see two main situations here. If tools are used by researchers *purely* to test new algorithms or ideas, *e.g.* a new visualization layout atop of the HEB implemented in SolidSX [38], then large investments in tool infrastructure (Sec. 2) is seen as waste. If tools are needed in case studies with external users or, even stronger, in real-life projects, then their usability is key to acceptance and success [5, 43, 39]. For example, valuable insight obtained from user-testing our toolset on large software projects with more than 70 students, 15 researchers, and 30 IT developers could never have been reached if our tools had not been mature enough for users to accept working with them in the first place. Hence, we believe that academic tools intended for other users than their immediate researcher creators should not compromise on *critical* usability (*e.g.* interactivity, scalability, robustness). However, effort required for *adoptability* (*e.g.* manuals, installers, how-to's, support of many input/output formats, rich GUIs), which is critical only in later deployment phases, can be limited.

6.2. *How to integrate and combine independently developed tools?*

This is an extremely challenging question as both the integration degree required and the tool heterogeneity vary widely in practice. For SVA tools, we have noticed the following practical patterns to provide good returns on investment, in increasing order of difficulty:

- *dataflow:* Tools communicate by reading and writing several data files in standardized formats, *e.g.* SQL for tables, GXL and XML (for attributed graphs) [44], and FAMIX and XMI (for design and architecture models) [45]. This easily allows creating dataflow-like tool pipelines, like the excellent Cpp2Xmi UML diagram extractor involving Columbus and Graphviz [46].

- *shared databases:* Tools communicate by reading and writing a single shared fact database which stores code, metrics, and relations, typically as a combination of text files (for code), XML (for lightweight structured data), and proprietary binary formats (for large datasets such as ASGs or execution traces). This is essentially the model used by Eclipse's CDT, Visual Studio's Intellisense, and SolidSX. As opposed to dataflows, shared databases support the much finer-grained data access required by interactive visualization tools *e.g.* for real-time browsing of dependencies (SolidSX) or symbol queries (Eclipse, Visual Studio, SolidFX). However, adding third-party tools to such a database requires writing potentially complex data converters.

- *common API:* Tools communicate by using a centralized API *e.g.* for accessing shared databases but also executing operations on-demand. This allows functionality reuse and customization at a finer-grained level of 'components' rather than monolithic tools. Although a common API does not necessarily enforce a shared tool implementation (code base), the former typically implies the latter in practice. API examples for SVA tools are the Eclipse, Visual Studio, and CodeCrawler [9] SDKs and, at a lower level, the Prefuse and Mondrian toolkits for constructing InfoVis applications [47, 48]. Our own toolset also offers a common C++ API for the table lens, treemap, and HEB visualizations which allowed us to easily embed these in a wide range of applications, see *e.g.* [27, 39, 40]. Common APIs are more flexible than shared databases, as they allow control *and* data flow composition. However, they are much harder to use in practice, as they impose coding restrictions and high learning costs (for tool builders) and costly maintenance (for API providers).

In practice, most SVA toolsets use the dataflow or shared database model which nicely balances costs with benefits. This is the case of our toolset (see Sec. 3) and also the conceptually similar SQuAVisiT toolset [49]. The main difference between SQuAVisiT and our toolset is the integration level: in our case, analysis and visualization are tighter integrated, *e.g.* the built-in static analysis in SolidSX, clone detection in SolidSDD, SolidSX integration in Visual Studio, and clone visualization (SolidSX used by SolidSDD), realized by a single fact database, the event-based mechanism described in Sec. 4.1.3, and the shared C++ visualization components API.

6.3. What are the lessons learned and pitfalls in building tools?

SVA tool building is mainly a design activity. Probably the most important element for success is striving to create visual and interaction models which optimally fit the 'mental map' of the targeted users. Within space limitations, we outline the following points:

- *2D vs 3D:* software engineers are used to 2D visualizations, so they will accept these much easier than 3D ones [50]. We found no single case when a 3D visualization was better accepted than a 2D one in our work. As such, we abandoned earlier work in 3D visualizations [18] and focused on 2D visualizations only.

- *interaction:* too much interaction and user interface options confuse even the most patient users. A good solution is to offer problem-specific minimalist interaction paths or wizards, and hide the complex options under an 'advanced' tab. This design is visible in the user interfaces of both SolidSX and SolidSDD.

- *scalable integration* of analysis with visualization is absolutely crucial for the acceptance of the latter [5, 42]. However, providing this is very costly. We estimate that over 50% of the entire code base of our toolset (over 700 KLOC) is dedicated to efficient data representation for smooth integration. For datasets up to a few hundred thousand items, a SQLite fact database performs very well (Sec. 3). However, heavyweight parsing, such as performed by SolidFX, creates much larger datasets (full ASGs of tens of millions of elements, roughly 10..15 elements per line of code). To efficiently store and query this, we opted to store such data in a custom binary format which minimizes space and maximizes search speed [25]. The same is true for plain source code, which is best stored as separate text files. The SQL database is still used as a 'master' component which points to such special storage schemes for particular datasets. Finally, we use XML mainly as a data interchange format for lightweight datasets, *e.g.* SolidSX accepts all its input either as SQL or XML.

However, our experience is that XML does not lend itself well for storing arbitrary relational data for large datasets and efficiently implementing complex queries.

6.4. What are effective techniques to improve the quality of academic tools?

For VA tools, quality strongly depends on usability. Research tools aimed at quickly testing new visual algorithms should maximize API simplicity; here, Prefuse and Mondrian are good examples. Tools aimed at real-world software engineering problems should maximize end-user effectiveness. For SVA tools, this is further reflected into uncluttered, scalable, and responsive displays, and tight integration for short analysis-visualization sensemaking loops. Recent InfoVis advances have significantly improved the first points. However, integration remains hard, especially given that the academic 'value model' give tool-related studies relatively lesser credit than technical papers.

6.5. Are there any useful tool building patterns for software engineering tools?

For SVA tools, we see the following elements as present in most such tools we are aware of:

- *architecture:* the dataflow and shared database models are probably the widest used composition patterns.
- *visualizations:* 2D visualizations using dense pixel layouts like the table lens, HEBs, treemaps, and annotated text offer high scalability and ease of use, so are suitable for large datasets created from static analysis; node-link layouts generate too much clutter for graphs over about 1000 nodes and/or edges, but are better when position and shape encode specific meaning, like for (UML) diagrams (see further 6.6). Shaded cushions, originally promoted by treemaps [29], are a simple to implement, fast, visually scalable, and effective instrument of conveying structure atop of complex layouts.
- *integration:* tightly integrating independently developed analysis and visualization tools is still an open challenge. Although the socket-based mechanism outlined in Sec. 4.1.3 has its limitations, *e.g.* it cannot communicate shared state, it strikes a good balance between simplicity and keeping the software stacks of the several tools to integrate independent.
- *heavyweight vs lightweight analysis:* Lightweight static analyzers are considerably simpler to implement, deploy, and use, deliver faster performance, and may produce sufficient information for visualization (see Sec. 4.1.1). However, once visualization requirements increase, as it typically happens with a successful tool, so do the requirements on its input analyzer. Extending static analyzers is, however, not a simple process, and typically requires switching to a completely new analyzer tool. As such, keeping the fact database model simple and weakly typed offers more flexibility in switching analysis (and visualization) tool combinations.

6.6. How to compare or benchmark such tools?

Benchmarking SVA tools can be done by lab, class, or field user studies, or using the tool in actual IT projects, either by comparing several tools against each other [40] or by comparing a tool with a predefined set of desirable requirements [15, 43]. Measuring *technical* aspects *e.g.* speed, visual scalability, or analysis accuracy can be done using de facto standard datasets in the SoftVis community, *e.g.* the Mozilla Firefox, Azureus, or JHotDraw source code bases, which have been used in the ACM SoftVis, IEEE Vissoft, and IEEE MSR conference 'challenges'. Measuring a SVA tool's end-to-end *usefulness* is harder as it depends on task and context specific details; 'insight' and 'comprehension' are hard to quantify. Still, side-by-side tool comparison can be used. For example, Figure 5 shows four SVA tools (Ispace, CodePro Analytix, SonarJ, and SolidSX) whose effectiveness in supporting an industrial corrective maintenance task we compared [51, 42]. This, and similar, studies confirmed our supposition that node-link layouts are effective in program comprehension only for relatively small graphs (up to a few hundred nodes), which is the main reason why our current toolset focuses on the more scalable HEB layout. Other useful instruments in gathering qualitative feedback are 'piggybacking' the tool atop of an accepted toolchain (*e.g.* Eclipse or Visual Studio) and using established community blogs for getting user sentiment and success (or failure) stories. From our experience, we noticed that this technique works for both academic and commercial tools.

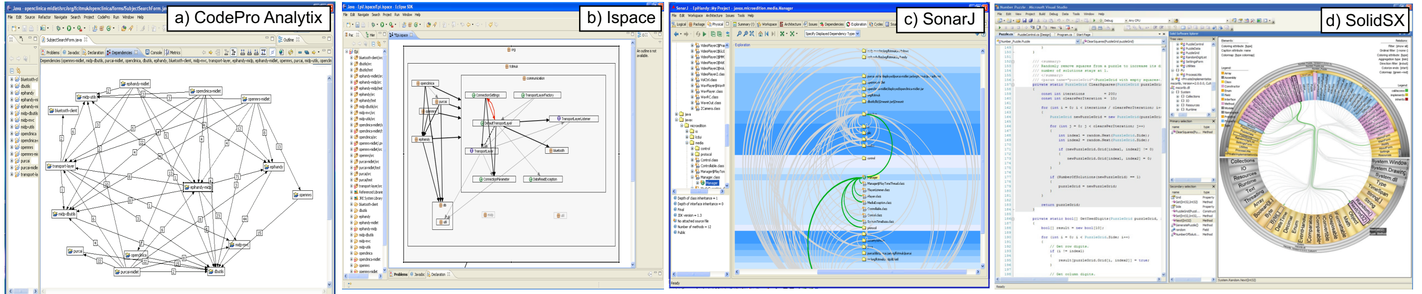


Figure 5: Four structure-and-dependency SVA tools compared for solving a corrective maintenance problem

6.7. What particular languages and paradigms are suited to build tools?

For SVA tools, our experience strongly converges to a minimal set of technologies, as follows:

- *graphics*: OpenGL, possibly augmented with simple pixel shader extensions, is by far the best solution in terms of portability, ease of coding, deployment, and performance; the same experience is shared by other researchers, *e.g.* Holten [28].
- *core:* scalability to millions of data items, relations, and attributes can only be achieved in programming languages like C, C++, or Delphi. Over 80% of all our analysis and application code is C++; Although Java is a good candidate, its performance and memory footprint are, from our experience, still not on par with compiled languages.
- *scripting:* Flexible configuration can be achieved in lightweight interpreted languages. The best candidate we found in terms of robustness, speed, portability, and ease of use was Python. Tcl/Tk (which we used earlier in [18] or Smalltalk (used by [9]) are also possible, but in our view require more effort for learning, deploying, and optimization.

7. Conclusions

In this paper, we have presented our experience in developing software visual analysis (SVA) tools, starting from research prototypes and ending with a commercial toolset. During this evolution and iterative design process, our toolset has converged from a wide variety of techniques to a relatively small set of proven concepts: the usage of a single shared weakly-typed fact database, implemented in SQL, which allows tool composition by means of shared fact selections; the usage of a small number of scalable InfoVis techniques such as table lenses, hierarchically bundled edge layouts, annotated text, timelines, and dense pixel charts; control flow composition by means of lightweight socket-based adapters as multiple linked-views in one or several independently developed tools; tool customization by means of Python scripts; and efficient core tool implementation using C/C++ and OpenGL.

We illustrated our toolset by means of two of its most recent applications: SolidSX for visualization of program structure, dependencies, and metrics, and SolidSDD for extraction and visualization of code clones. We outlined the added value of combining several tools in typical visual analysis scenarios by means of simple examples and a more complex industrial post-mortem software assessment case. Finally, from the experience gained in this development process, we addressed several questions relevant to the wider audience of academic tool builders.

Ongoing work targets the extension of our toolset at several levels: New static analyzers to support `gcc` binary analysis and lightweight zero-configuration C/C++ parsing; dynamic analysis for code coverage and execution metrics; and integration with the Eclipse IDE. Finally, we consider extending our visualizations with new metaphors which allow an easier navigation from source code to structural level by combining HEB layouts and annotated code text in a single scalable view.

References

[1] T. A. Standish, An Essay on Software Reuse, IEEE TSE 10 (5) (1984) 494–497.
[2] T. Corbi, Program Understanding: Challenge for the 1990s, IBM Systems Journal 28 (2) (1999) 294–306.
[3] S. Reiss, The paradox of software visualization, in: Proc. IEEE Vissoft, 59–63, 2005.

[4] S. Charters, N. Thomas, M. Munro, The end of the line for Software Visualisation?, in: Proc. IEEE Vissoft, 27–35, 2003.
[5] R. Koschke, Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey, J. Soft. Maint. and Evol. 15 (2) (2003) 87–109.
[6] P. C. Wong, J. J. Thomas, Visual Analytics, IEEE CG&A 24 (5) (2004) 20–21.
[7] J. J. Thomas, K. A. Cook, Illuminating the Path: The Research and Development Agenda for Visual Analytics, National Visualization and Analytics Center, 2005.
[8] M. van den Brand, J. Heering, P. Klint, P. Olivier, Compiling language definitions: the ASF+SDF compiler, ACM TOPLAS 24 (4) (2002) 334–368.
[9] M. Lanza, *CodeCrawler* - Polymetric Views in Action, in: Proc. ASE, 394–395, 2004.
[10] F. Boerboom, A. Janssen, Fact Extraction, Querying and Visualization of Large C++ Code Bases, in: MSc thesis, Faculty of Math. and Computer Science, Eindhoven Univ. of Technology, 2006.
[11] T. Mens, S. Demeyer, Software Evolution, Springer, 2008.
[12] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice, Springer, 2006.
[13] N. Fenton, S. Pfleeger, Software Metrics: A Rigorous and Pracical Approach, Chapman & Hall, 1998.
[14] S. Diehl, Software Visualization Visualizing the Structure, Behaviour, and Evolution of Software, Springer, 2007.
[15] H. Kienle, H. A. Müller, Requirements of Software Visualization Tools: A Literature Survey, in: Proc. IEEE Vissoft, 92–100, 2007.
[16] SolidSource BV, SolidSX, SolidSDD, SolidSTA, and SolidFX tool distributions, `www.solidsourceit.com`, 2010.
[17] SVCG, Scientific Visualization and Computer Graphics Group, Univ. of Groningen, Software Visualization and Analysis, `www.cs.rug.nl/svcg/SoftVis`, 2010.
[18] A. Telea, A. Maccari, C. Riva, An Open Toolkit for Prototyping Reverse Engineering Visualizations, in: Proc. Data Visualization (IEEE VisSym), IEEE, 67–75, 2002.
[19] G. Lommerse, F. Nossin, L. Voinea, A. Telea, The *Visual Code Navigator*: An Interactive Toolset for Source Code Investigation, in: Proc. InfoVis, IEEE, 24–31, 2005.
[20] L. Voinea, A. Telea, J. J. van Wijk, CVSscan: visualization of code evolution, in: Proc. ACM SOFTVIS, 47–56, 2005.
[21] L. Voinea, A. Telea, Visual Querying and Analysis of Large Software Repositories, Empirical Software Engineering 14 (3) (2009) 316–340.
[22] M. Termeer, C. Lange, A. Telea, M. Chaudron, Visual exploration of combined architectural and metric information, in: Proc. IEEE Vissoft, 21–26, 2005.
[23] S. Moreta, A. Telea, Multiscale Visualization of Dynamic Software Logs, in: Proc. EuroVis, 11–18, 2007.
[24] A. Telea, L. Voinea, Visual Software Analytics for the Build Optimization of Large-scale Software Systems, Computational Statistics (in print), see also `www.cs.rug.nl/~alext/PAPERS`.
[25] A. Telea, L. Voinea, An Interactive Reverse-Engineering Environment for Large-Scale C++ Code, in: Proc. ACM SOFTVIS, 67–76, 2008.
[26] R. Rao, S. Card, The table lens: Merging graphical and symbolic representations in an interactive focus+context visualization for tabular information, in: Proc. CHI, ACM, 222–230, 1994.
[27] A. Telea, Combining extended table lens and treemap techniques for visualizing tabular data, in: Proc. EuroVis, 51–58, 2006.
[28] D. Holten, Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data, in: Proc. IEEE InfoVis, 741–748, 2006.
[29] B. Shneiderman, Treemaps for space-constrained visualization of hierarchies, `www.cs.umd.edu/hcil/treemap-history`, 2010.
[30] A. Ludwig, Recoder Java analyzer, `recoder.sourceforge.net`, 2010.
[31] Redgate Inc., Reflector .NET API, `www.red-gate.com/products/reflector`, 2010.
[32] R. Ferenc, A. Beszédes, M. Tarkiainen, T. Gyimóthy, Columbus – Reverse Engineering Tool and Schema for C++, in: Proc. ICSM, IEEE, 172–181, 2002.
[33] LLVM Team, Clang C/C++ analyzer home page, `clang.llvm.org`, 2010.
[34] Y. Lin, R. C. Holt, A. J. Malton, Completeness of a Fact Extractor, in: Proc. WCRE, IEEE, 196–204, 2003.
[35] Bell Labs, CScope, cscope.sourceforge.net, 2007.
[36] T. Kamiya, CCfinder clone detector home page, `www.ccfinder.net`, 2010.
[37] VTK Team, The Visualization Toolkit (VTK) home page, `www.vtk.org`, 2010.
[38] A. Telea, O. Ersoy, Image-based Edge Bundles: Simplified Visualization of Large Graphs, Comp. Graph. Forum 29 (3) (2010) 65–74.
[39] A. Telea, L. Voinea, A Tool for Optimizing the Build Performance of Large Software Code Bases, in: Proc. IEEE CSMR, 153–156, 2008.
[40] L. Voinea, A. Telea, Case Study: Visual Analytics in Software Product Assessments, in: Proc. IEEE Vissoft, 57–45, 2009.
[41] M. Poppendieck, T. Poppendieck, Lean Software Development: An Agile Toolkit for Software Development Managers, Addison-Wesley, 2006.
[42] A. Telea, L. Voinea, O. Ersoy, Visual Analytics in Software Maintenance: Challenges and Opportunities, in: Proc. EuroVAST, Eurographics, 65–70, 2010.
[43] M. Sensalire, P. Ogao, A. Telea, Classifying desirable features of software visualization tools for corrective maintenance, in: Proc. ACM SOFTVIS, 87–90, 2008.
[44] R. Holt, A. Winter, A. Schurr, GXL: Towards a standard Exchange Format, in: Proc. WCRE, 162–171, 2000.
[45] S. Tichelaar, S. Ducasse, S. Demeyer, FAMIX and XMI, in: Proc. WCRE, 296–300, 2000.
[46] E. Korshunova, M. Petkovic, M. van den Brand, M. Mousavi, Cpp2XMI: Reverse Engineering for UML Class, Sequence and Activity Diagrams from C++ Source Code, in: Proc. WCRE, 297–298, 2006.
[47] Prefuse, The Prefuse Information Visualization Toolkit, `prefuse.org`, 2010.
[48] A. Lienhardt, A. Kuhn, O. Greevy, Rapid Prototyping of Visualizations using Mondrian, in: Proc. IEEE Vissoft, 67–70, 2007.
[49] M. van den Brand, S. Roubtsov, A. Serebrenik, SQuAVisiT: A Flexible Tool for Visual Software Analytics, in: Proc. CSMR, 331–332, 2009.
[50] A. Teyseyre, M. Campo, An Overview of 3D Software Visualization, IEEE TVCG 15 (1) (2009) 87–105.
[51] M. Sensalire, P. Ogao, A. Telea, Model-Based Analysis of Adoption Factors for Software Visualization Tools in Corrective Maintenance, Univ. of Groningen, the Netherlands, Tech. Report SVCG-RUG-10-2010, `www.cs.rug.nl/~alext/PAPERS/Sen10.pdf`, 2010.