The Solid* Toolset for Software Visual Analytics of Program Structure and Metrics Comprehension: From Research Prototype to Product

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 fact extraction with information visualization to support program comprehension. However, building efficient and effective SVA tools is highly challenging, as it involves extensive 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 toolset's usage for constructing software visualizations with examples in education, research, and industrial contexts. We discuss the design evolution from research prototypes to integrated, scalable, and easy-to-use products, and present several guidelines for the development of efficient and effective SVA 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 dataset sizes, long learning curves, and poor integration with software analysis or development toolchains, as strongly voiced by several researchers [3, 4, 5, 6].

Visual analytics (VA) integrates graphics, visualization, interaction, and data collection and analysis to support reasoning and sensemaking for complex problem solving in engineering, finances, security, and geosciences [7, 8]. These fields share many similarities with software maintenance in terms of *data* (large databases, structured text, and graphs), *tasks* (sensemaking by hypothesis creation, refinement, and validation), and *tools* (combined analysis and visualization). VA stresses tool integration, as opposed to pure data mining or fact extraction (whose main focus is scalability) or information visualization (Infovis, mainly focused on presentation). As such, VA is a promising model for building effective and efficient software visual analysis (SVA) tools. However, building SVA tools for software comprehension is particularly challenging, as developers have to master static analysis, fact extraction, graphics, information visualization, and user interaction design technologies.

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 used by many end-users in the IT industry. Our toolset supports static analysis, quality metrics computation, clone detection, and state-of-the-art Infovis techniques such as table lenses, bundled graph layouts, cushion treemaps, and dense pixel charts. The toolset addresses several use-cases, of which we focus here on two: visual analysis of program structure and code duplication. These use-cases can be combined to support tasks such as assessing system quality and planning refactoring.

The contributions of this paper are as follows:

*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)

- describe our toolset with respect to the ease of installation, usage, and applicability to program comprehension;
- detail 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 research and industrial contexts, with focus on efficiency, effectiveness, acceptance, and experienced pitfalls;
- present evidence for our design decisions based on actual toolset usage.

Section 2 introduces software visual analytics. Section 3 details our toolset's architecture. Section 4 details two tools in our toolset which offer fact extraction and visualization of software structure, metrics, and code duplicates, and outlines its installation, usage, and extensibility. Section 5 shows the use of our toolset in an industrial software assessment case. Section 6 discusses the lessons learned in developing efficient, effective, and accepted SVA tools. Finally, section 7 concludes the paper.

2. Related Work

Software visual analytics can be roughly divided into fact extraction and visualization, as follows.

Fact extraction covers the gathering of information from source code, binaries, and source control management (SCM) systems such as CVS, Subversion, Git, CM/Synergy, or ClearCase. Raw data delivered by syntax and semantic analysis is refined into call and control flow graphs, program slices, code duplicates (clones), software quality metrics (*e.g.* 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 trade fact completeness and accuracy for speed and simplicity; and *heavyweight* ones, which do full syntactic and semantic analysis at higher cost. Well-known static analyzers include LLVM [9], ROSE [10], Cppx [11], Columbus [12], Eclipse CDT [13], and Elsa [14] (for C/C++), Recoder [15] (for Java), and ASF+SDF (a meta-framework with language-specific front-ends) [16]. Metric tools include CodeCrawler [17], Understand [18], and Visual Studio Team System (VSTS). An overview of C/C++ static analysis tools is given by Boerboom and Jassen [19]. Insights in software evolution and software quality metrics are given by Mens *et al.* [20], Lanza *et al.* [21], and Fenton *et al.* [22].

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 by Diehl [23]. Well-known software visualization systems include Rigi [24], VCG [25], aiSee [26], Mondrian [27], and sv3D [28] (for structure and metrics); and CodeCity [29, 30] and SeeSoft [31] (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, ICSE, 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, 3, 32, 33]. Combining analysis and visualization in one 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. SVA Program Comprehension 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 [34]. Table 1 outlines our most important tools, with binaries and source code available [35]. Evolution of our toolset highlights several aspects relevant to the path of effective academic-to-widely-used tool development: architecture and design decisions, choice of techniques and software components, and effective presentation aspects. We next detail the data and visualization architecture of our toolset (Secs. 3.1 and 3.2). The toolset's architecture evolution from research prototypes to products is discussed further in Sec. 6.9.

Tool	Targeted data types	Visual techniques	Analysis techniques	Drawing	Data storage	Users
SoftVision (2002) [36]	software architecture	node-link layouts (2D and 3D)	none (visualization tool only)	Open Inventor	text files (RSF format)	10..20
CSV (2004) [37]	source code syntax and metrics	pixel text, cushions	C++ static analysis (`gcc` based parser)	Open Inventor	plain text and XML	10..20
CVSscan[38] (2005)	file evolution	dense pixel charts annotated text	CVS fact extraction (authors & line-level changes)	OpenGL	text files and SQLite	20..30
CVSgrab (2006) [39]	project evolution	dense pixel charts, cushions	CVS/SVN fact extraction (project-level changes)	OpenGL	text files and SQLite	30..50
MetricView (2006) [40]	UML diagrams and quality metrics	2D node-link layouts, table lenses	C++ lightweight static analysis (class diagram extraction)	Open Inventor	UML files (XMI format)	50..80
MemoView (2007) [41]	dynamic logs (memory allocations)	table lenses, cushions, timelines	C runtime instrumentation (`libc` malloc/free logging)	OpenGL	binary files (own format)	5..10
SolidBA (2007) [42]	build dependencies build cost metrics	table lenses, 2D node-link layouts	C++ dependency mining, and automated refactoring	OpenGL	SQLite	15..25
SolidFX (2008) [43]	reverse engineering	pixel text, annotations, table lenses	C/C++ heavyweight static analysis	OpenGL	binary files and SQLite	50..75
SolidSTA (2008) [34]	file and project-level evolution	dense pixel charts, cushions, timelines	CVS/SVN/Git fact extraction and source code metrics	OpenGL	SQLite	200..250
SolidSX (2009) [34]	structure, metrics, associations	HEB views, treemaps, table lenses, cushions	.NET, C++, Java lightweight static analysis	OpenGL	SQLite	200..250
SolidSDD (2010) [34]	code duplicates, structure, metrics	HEB views, treemaps, table lenses, pixel text	C, C++, Java, C# configurable syntax-aware clone detection	OpenGL	SQLite	100..150

Table 1: Software visual analytics tools - evolution history (Sec. 6.9). Tools discussed in this paper are in bold. The first six tools are research prototypes. The latter five tools (Solid*) are commercial products.

3.1. Data architecture

Our toolset uses a simple dataflow architecture (Fig. 1). Raw input data comes as unversioned source code or versioned files stored in SCM systems. From raw data, we extract several *facts*: syntactic and semantic structure, static dependency graphs, and source code duplication. The data architecture used to manage these facts is detailed next.

Relational and attribute data is stored into a SQLite database [44] whose entries point to flat text files (for actual source code) and binary files (for complete syntax trees, see 4.2.1). Fact extraction is implemented by specific tools: parsers and semantic analyzers for source code, and clone detectors for code duplication (see Sec. 4).

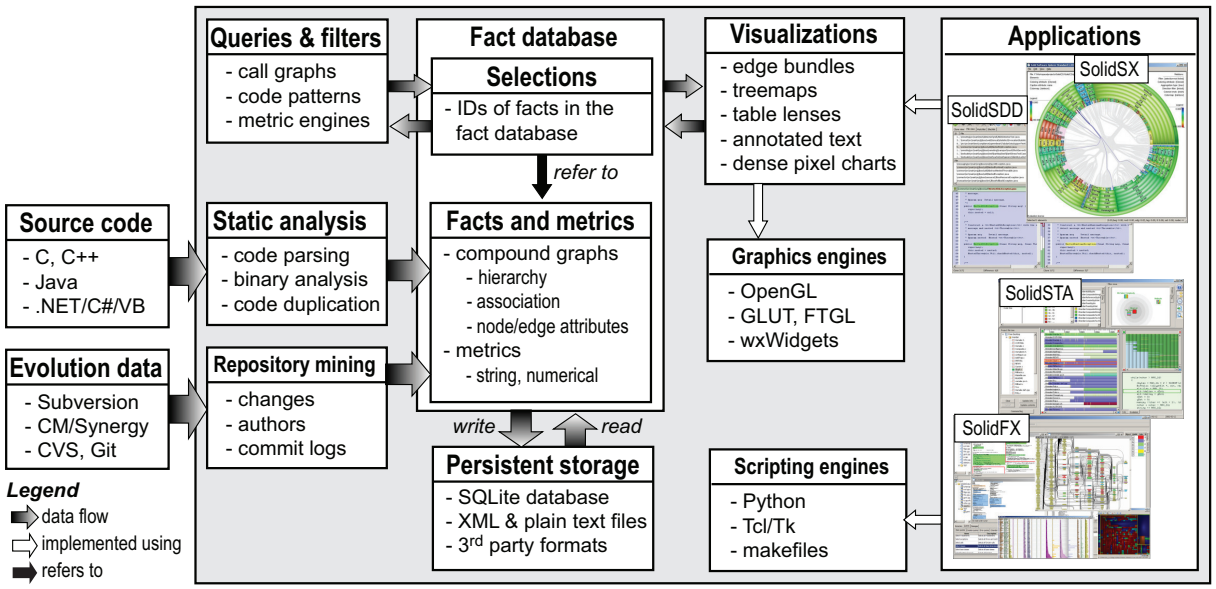

Figure 1: Toolset architecture (see Section 3).

Besides facts, our database stores two other elements: selections and metrics. *Selections* are sets of facts or other selections.

They support the iterative data refinement in the so-called visual sensemaking cycle of VA [7, 8]. Selections are created either by tools, *e.g.* extraction of class hierarchies or call graphs from annotated syntax graphs (ASGs), or interactively by users. Selections have unique names by which they are referred by the tools and under which they are persistently saved. *Metrics* are numerical, ordinal, categorical, or text attributes. Metrics can be added to facts or selections by tools, *e.g.* complexity, fan-in, fan-out, cohesion, and coupling, or set as annotations by users, *e.g.* marking certain classes in a UML view as being 'unsafe'.

Using an SQL database to query large attributed relational data can pose efficiency problems if this requires multiple-table joins [45, 46]. To alleviate this, we adopted the following schema (see Fig. 2):

- each node, containment (hierarchy) or association edge, and selection, has a unique ID;
- a *hierarchy* table: each row stores a containment edge, listed as (parent, child) node IDs;
- an *association* table: each row stores an association edge, listed as (from, to) node IDs;
- a *node attribute* table: each row stores all attributes (metrics) $a_1, \ldots, a_n$ of a given node as n columns;
- an *edge attribute* table: each row stores all attributes (metrics) $a_1, \ldots, a_n$ of a given edge as n columns; different edge types, *e.g.* calls, uses, includes, are modeled by adding an edge-type attribute;
- two *selection* tables per selection, for the node IDs and edge IDs of the selected facts, respectively; additional attributes (metrics) of the selected facts in the context of a specific selection can be added as extra columns.

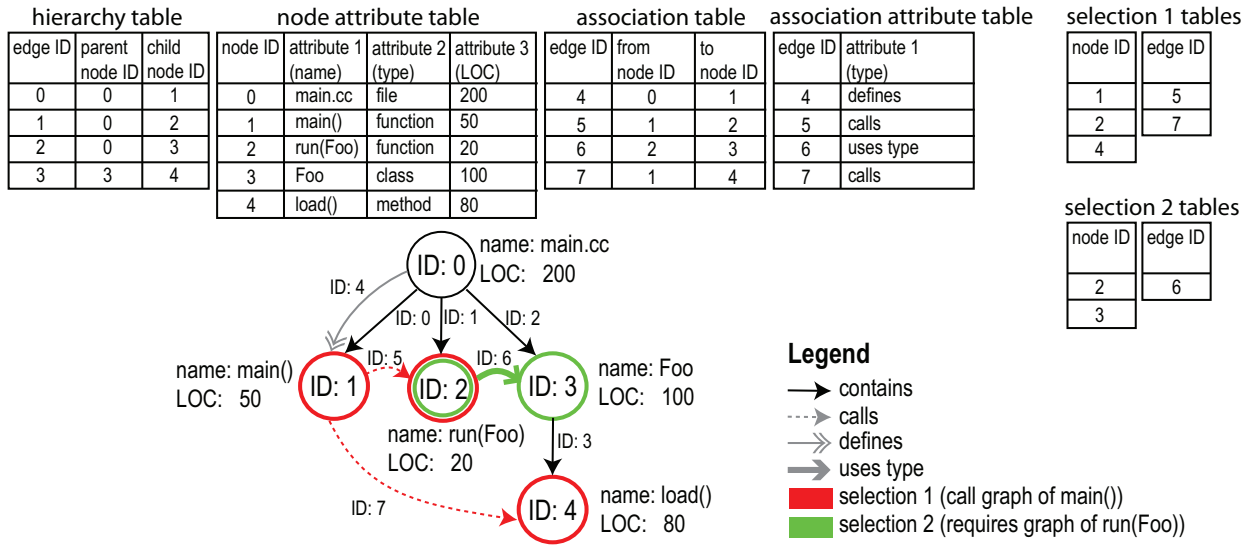

Figure 2: Database schema (top) for a compound attributed graph (bottom) and two selections: the call graph of main() and the 'requires' graph of run(Foo).

This schema can store any compound (hierarchy-and-association) attributed graph: ASGs, class hierarchies, call graphs, or clone relations. New association types can be added to the database without changing its schema, since types are stored as attributes. The example in Fig. 2 (bottom) shows this for a simple program. Hierarchy consists of a file *main.cc* with two functions *main*() and *run*(*Foo*), and a class *Foo* with a method *load*(). Associations capture call; define; and 'uses type' relations (*e.g.* the fact that *run*(*Foo*) uses the type *Foo*), modeled as edge 'type' attributes. Nodes have two attributes: name and lines-of-code size (LOC). Two selections exist: the call graph of *main*() (red), and the 'requires' graph of *run*(*Foo*) (green)[1].

On-the-fly computed data, *e.g.* selections, metrics, visualization and layout properties, and annotations, imply the creation and/or editing of additional selection tables or attribute columns. Using a separate table for each selection optimizes speed

[1] We recommend viewing this paper in full color

and memory consumption, as in a typical analysis scenario tens or even hundreds of different selections of variable size are created and deleted dynamically. Missing values are naturally accommodated by the SQL database. Storing hierarchy data as a separate table from the associations enables efficient traversals of the compound graph for *e.g.* level-of-detail rendering and interactive selection. For instance, finding the children of a node uses just the hierarchy table. If we had containment and associations edges in the same table, distinguished *e.g.* by a special 'type' attribute in the association attribute table, finding children would need to query *both* the association table and association attribute table. Rendering and selection have to be as fast as possible to maximize visual fluency, so we opted for a separate hierarchy table solution. The same special treatment of hierarchy relations is used also by other graphics or graph layout toolkits, *e.g.* OpenInventor [47] and Graphviz [48].

The above simple model scales well to fact databases of hundreds of thousands of facts with tens of metrics per fact [49]. Multiple hierarchies can be added as multiple hierarchy tables. Simple queries and filters can be directly written in SQL. Structural queries, *e.g.* connected components or reachability, are efficient, as they require just iterating over the node and association tables. For example, rendering the graph in Fig. 3 (4K nodes, 15K associations) takes subsecond time on a commodity PC.

Selections are the glue that allows composing different tools. All analysis (*e.g.* filters, queries, or transformations) and visualization components in our toolkit read selections, and optionally create selections (see Fig. 1). Hence, selections are the only interface that tools use to communicate with each other. In this way, existing or new tools can be composed either statically or at run-time with no configuration costs. To ensure consistency, each tool decides internally whether (and how) to execute its function on a given input selection.

3.2. Visualization architecture

Visualizations display selections and allow users to interactively navigate, pick elements, and customize the visual aspect. Since they only receive selection names as inputs, they 'pull' their required referred data on demand, *e.g.* a source code viewer opens the files in its input selection to render the text. Tools can freely decide to cache data internally to reduce traffic with the fact database, if desired. This decision is completely transparent at the architecture level.

Our current toolset offers several visualizations, as follows. *Table lenses* show large tables by zooming out the table layout and drawing cells as pixel bars scaled and colored by data values [50]. Subpixel sampling techniques allow rendering tables up to hundreds of thousands of rows on a single screen [51, 35]. *Hierarchically bundled edges* (HEBs) compactly show compound (structure and association) software graphs, *e.g.* containment and call relations, by bundling association edges along the structure [52]. *Squarified cushion treemaps* show software structure and metrics for up to tens of thousands of elements on a single screen [53, 35]. We also provide classical views: metric-annotated code text, tree browsers, customizable color maps, legends, annotations, timelines, details-on-demand (tooltips), and text-based search tools. The above visualizations are illustrated by the two tools described next in Sec. 4.

A single treemap, HEB, or table lens can show the correlation of just a few attributes. We augment this by the *multiple correlated views* technique. Besides the 'input selection', which contains the data to render, each view has a 'user selection', which holds the data interactively selected in that view. Views that share input selections show the same data. Views sharing user selections highlight user-picked data in different contexts. All components are synchronized by an Observer pattern on selections; when data is modified by user interaction or by analysis engines, all toolset components update automatically.

4. Toolset Highlights: SolidSX and SolidSDD

We next present our toolset installation (Sec 4.1) and describe two tools of the toolset: the SolidSX structure analyzer (Sec. 4.2) and the SolidSDD clone analyzer (Sec. 4.3).

4.1. Toolset Installation and First Usage Steps

SolidSX and SolidSDD are provided as self-contained installers, freely available for research [34] on Windows XP and later editions. Installation takes a few minutes and mouse clicks, once an install location on the host system is provided. SolidSDD, which uses SolidSX internally (Sec. 4.3), installs SolidSX or uses a previously installed copy of SolidSX. No scripting, third-party package installation, environment setting, or compilation are needed. Manuals and sample datasets are also installed.

After installation, starting to use both tools is simple. To explore software structure and metrics with SolidSX, one only needs to load *.bsc* symbol files (for Visual C++), the root of a code base (for Java code), or any .NET assemblies (for C# or VB). An example tutorial of using SolidSX is provided separately [54]. To explore code clones with SolidSDD, one needs to

load the root of a source code base (C, C++, Java, or C#) and click the clone detection button. For example, the scenario in Sec. 4.3 can be replicated as follows: (1) install SolidSDD; (2) download the source code to analyze [55]; (3) point SolidSDD to the root of the downloaded code; (4) start the clone detection; (5) visualize the detected clones.

4.2. SolidSX: Structural Analysis

The SolidSX (Software eXplorer) tool supports the analysis of software structure, dependencies, and metrics. Several built-in parsers are provided: Recoder for Java source and bytecode [15], Reflector for .NET assemblies [56], and Microsoft's free parser for Visual C++ *.bsc* symbol files. Built-in filters refine parsed data into a compound attributed graph with folder-file-class-method and namespace-class-method hierarchies; calls, symbol usage, inheritance, and package or header inclusion dependencies; and basic metrics, *e.g.* code and comment size, complexity, fan-in, fan-out, and symbol source code location.

4.2.1. Static Analysis: Ease of Use Considerations

For .NET/VB/C#, Java, and Visual C++, static analysis is completely automated. The user only needs to input a root directory for code and, for Java, optional class paths. For Java, Recoder, a relatively less known analyzer [15], is close to ideal, as it delivers heavyweight information at 100 KLOC/second on a typical PC computer. For .NET, the Reflector lightweight analyzer is fast, robust, and simple to use. The same holds for Microsoft's *.bsc* symbol file parser.

C/C++ static analysis beyond Visual Studio is much harder. Setting up C/C++ analysis without a tight analyzer integration with a build system is complex and time consuming. Language dialect, files to analyze, include and library paths, facts to export, and handling of analysis errors must be explicitly specified if there is no build system, *e.g.* project file or makefile, to take these from. An example hereof is the integration of SolidSX with our separate SolidFX C/C++ static analyzer [43]. SolidFX scales to millions of lines of code, covers several dialects (*e.g.* gcc, C89/99, ANSI C++), handles incorrect and incomplete code, has a preprocessor, and integrates with the `gcc` and Visual C++ build systems via compiler wrapping [12]. Still, certain options such as platform defines and headers cannot be inferred from build systems and must be manually specified. Also, compiler wrapping requires a working build system on the target machine, which is not always the case. Other heavyweight C/C++ analyzers *e.g.* Columbus [12] or Clang [9] have the same issues. We also considered using lightweight C/C++ analyzers, *e.g.* CPPX [11], gccxml, and MC++. We found that these deliver massively incorrect information, due to simplified preprocessing and name lookup implementations. Finally, we considered using the built-in C/C++ parsers of Eclipse CDT, KDevelop, QtCreator, and Cscope [57]. While better in correctness, such parsers depend in complex ways on their host IDEs and do not have well-documented APIs, so cannot be embedded into third-party tools. Extended discussions with Roberto Raggi, the creator of KDevelop and QtCreator, confirmed this point.

4.2.2. Structure Visualization

SolidSX offers four views (Fig. 3): tree browsers, table lenses of node and edge metrics, treemaps, and HEB layouts [52]. All views have carefully designed *presets* which allow using them with no extra customization. All views show selections from the fact database created by static analysis (Sec. 3.2). User selections, created interactively or by queries, effectively 'link' facts shown in different views, which enables one to easily correlate structure, dependencies, and metrics along different viewpoints.

Figure 3 illustrates this on a C# system of around 45 KLOC (a graphical number puzzle program, whose source code is available in the tool distribution). The radial HEB view shows function calls over system structure, with caller edge ends blue and callee edge ends gray. Node colors show McCabe's complexity on a green-to-red colormap. We can now correlate complexity with system structure: We see that the most complex functions (warm colors) are in the module and classes top-left in the HEB view. The table lens shows several function-level code metrics, sorted on decreasing complexity. This shows how different metrics correlate with each other. Linked HEB-table lens selections support queries such as "what are the metrics of this module?" or "where are the most complex functions located?" 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' shows that complexity correlates well with size. Constructing the entire scenario, including the static analysis, takes about 2 minutes and under 20 mouse clicks.

SolidSX brings several visual additions atop of Holten's HEB layout [52]. Luminance textures, added to nodes, enhance the hierarchical structure. When nodes are collapsed and/or expanded, the layout is smoothly animated between the initial and final views, which reduces the visual change and helps users maintain their mental map (for an illustration, see the actual tool in use). Multiple edges between collapsed nodes are replaced by a single edge. This edge's color shows the aggregated value

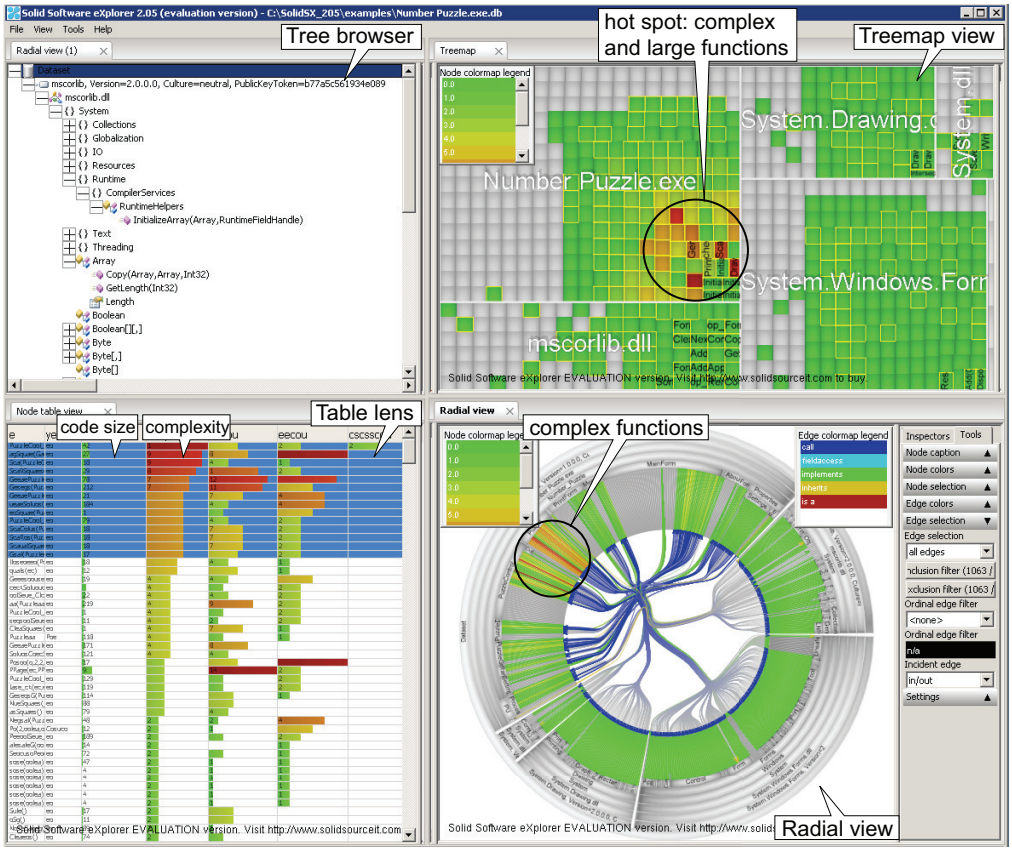

Figure 3: SolidSX views (tree browser, treemap, table lens, radial HEB).

(min, max, or average) of the collapsed edges' attributes, as indicated by user preferences. Adjacent nodes which are under a few pixels in width are replaced by gray, untextured bars. This indicates that the view cannot show the full dataset and also maintains a high frame-rate regardless of dataset size, since the number of rendered nodes never exceeds the actual view size divided by the minimal node size. A similar technique is used for rendering cells in the table view [51]. Finally, attribute-based color mapping allows quickly locating elements of interest in a large visualization, such as highly complex elements.

4.2.3. Toolchain Integration

Similarly to Koschke [5], we noticed that *integration* in accepted toolchains is a key acceptance factor when using our tools in industrial applications [42, 49, 58, 43]. We address integration by a *listener* mechanism. The SVA tool listens for command events sent asynchronously as Windows system messages, *e.g.* load a dataset, zoom on some subset, change view parameters, and also sends user interaction events (*e.g.* user has selected some data. This allows integrating SolidSX in any third-party tool(chain) by building thin wrappers which read, write, and process such events. No changes to our tool's code are needed. For example, we integrated SolidSX in Visual Studio by writing a thin plug-in of around 200 LOC which translates between the IDE and SolidSX events. Selecting and browsing code in the two tools is now in sync. The open SQLite database format further simplifies data integration.

4.3. SolidSDD: Clone Inspection

Code duplication (or clone) detection is an important tool in software maintenance. Although many clone detectors exist, few show the clone information in easy to understand ways. Simple clone lists do not show how clones are spread over a system's structure [59]. Node-link clone views inherit the limited scalability and visual clutter of force-directed graph layouts [60].

Adjacency matrices, showing clones between file pairs [61], are not immediately intuitive for software engineers, and do not show clone relations at several levels-of-detail, *e.g.* function, file, and folder.

To address these visualization issues, we developed SolidSDD (Software Duplication Detector). Clone detection uses the same algorithm as CCfinder [62], 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. From clones detected in an input C, C++, Java, or C# code base, we store a *compound duplication graph* in our fact database. Nodes are cloned code fragments. Edges indicate clone relations. Hierarchy is added either from the code directory structure or from a user-supplied dataset (*e.g.* from static analysis). Nodes and edges have metrics, *e.g.* percentage of cloned code, number of distinct clones, and whether a clone includes identifier renaming or not. Metrics are aggregated bottom-up using the hierarchy information (see Sec. 4.2.2).

We use the compound graph to visualize clones with two different views, as follows (see Fig. 4). Our test dataset is the well-known Visualization Toolkit code base [55]. On VTK version 5.4 (2420 C++ files, 668 C files, 2660 headers, 2.1 MLOC in total), SolidSDD found 946 clones in 280 seconds on a 3 GHz PC with 4 GB RAM, using the default tool settings. The first view (Fig. 4 a,b,e) is the SolidSX tool described in Sec. 4.2. Figure 4 a shows the code clones atop of the system structure. Edges show aggregated clone relations between files: two files are connected when they share at least one clone. Node colors show the percentage of cloned code in a subsystem on a green-to-red (0..100%) colormap. Edge colors show percentage of cloned code in the clones represented by an edge. Figure 4 a shows that the VTK system has many intra-system clones (edges connecting files in the same folder) but also some inter-system clones (edge connecting files in different folders).

Three subsystems have high clone percentages (red tints in Fig. 4 a): *examples* (S_1), *bin* (S_2) and *Filtering* (S_3). Browsing these files we saw that clones in *examples* and *bin* are in tutorial code and test drivers, arguably created by copy-and-paste. Clones in *Filtering*, a core subsystem of VTK, are more interesting. In Fig. 4 b, we zoom on *Filtering* and select a file f (marked in black) which has over 50% cloned code: vtkGenericDataSetAlgorithm. This highlights the files f_c with which f shares clones, called the *clone partners* of f. We find five clone partners of f in the same *Filtering* subsystem (vtkStructured-GridAlgorithm, vtkDataObjectalgorithm, vtkUnstructuredGridAlgorithm, vtkHyperOctreeAlgorithm, vtkPolyDataAlgorithm, light blue in Fig. 4 b), and one in the *Rendering* subsystem (g, vtkLabelHierarchyAlgorithm, purple). Each such files contains a separate class, as standard in VTK. When writing these, developers probably copied and pasted code between sibling classes. Given VTK's guidelines to keep its subsystems independent *and* maximize code reuse, the clone g could be refactored by moving the common algorithm part to a superclass.

Figure 4 c shows the *text view* of SolidSDD. The top light-blue table shows all files with percentage of cloned code, number of clones, and presence of identifier renaming. Sorting this table allows *e.g.* finding files with the most clones or highest cloned code percentage. This view is linked with SolidSX via the message mechanism (Sec. 4.2.3), so selecting the file f in the HEB view (Fig. 4 c, black) highlights it in this table (in red). The table below (Fig. 4 c, middle panel) shows the clone partner files f_c of f. Here we find the file g which shares clones with f but is in another subsystem. We select g and use the two text views (Fig. 4 b, bottom panels) to study 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 synchronized to easily compare corresponding code fragments. Text is color-coded: non-cloned code (white), code in f which is cloned in g (light blue), renamed identifier pairs (green in left view, yellow in right view), and clone code in f whose clones are in some other file $h \neq g$ (light brown). The last color allows us to navigate from f to other clone partners h: Clicking on light brown code in the left view (f) in Fig. 4 c replaces the file g in the right view by the clone partner h, and also selects h in the clone partner list view. Fig. 4 d shows this. We now notice that code in f which is part of the clone $f - g$ (light blue in Fig. 4 c) is included in the clone $f - h$ (light blue in Fig. 4 c).

5. Toolset Applications

We have used our SVA toolset in both research [63, 64] and the industry [42, 65, 43]. We next describe several such use-cases and highlight strengths and weaknesses of our proposal.

5.1. Toolset Usage in Education

We used our SVA toolset in academic courses in two different contexts, as follows.

Tool end-user context: In the first case, 3^{rd} year BSc students at the University of Groningen, the Netherlands, used SolidSX for the course Software Quality Assurance and Testing (SQAT) [54] (30 students per year on average during 2008-2011) to

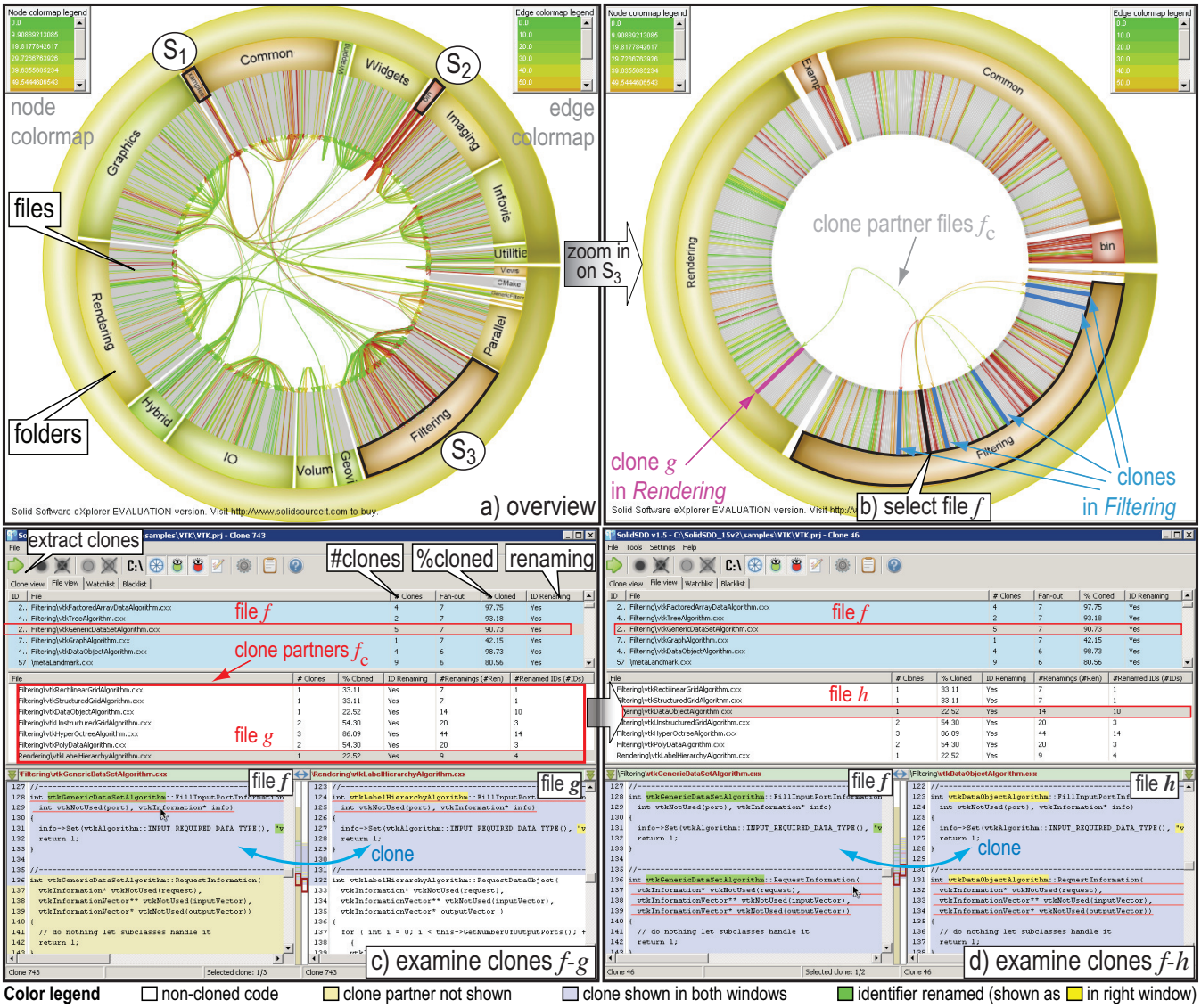

Figure 4: SolidSDD clone visualization using the HEB view (top) and text view (bottom) (see Sec. 4.3).

explore a 3500 LOC C++ image processing program [66, 67] in order to design test cases using white-box testing and also assess the program's modularity, complexity, portability, and testability. Facts were provided by the Visual C++ .bsc parser (Sec. 4.2). No programming was required. After an 8-week course, students worked one month on the assignment. Additionally, the students could use the Visual Studio Team System (VSTS) tool to explore the code. Feedback gathered from mandatory course evaluations showed that the students found the installation, learning, and usage or SolidSX very easy, although they had no previous experience in static analysis or software visualization. Table 2 shows tool-related points from this course evaluation, ranked on a 5-point scale (very limited, limited, neutral, good, very good), averaged for 87 students. The listed positive and/or negative observations aggregate comments given under a free-text heading concerning additional comments. The HEB view was found effective for assessing the modularity of a previously unknown code base, which replicates previous similar findings [52, 63]. On the negative side, they noted that SolidSX works on a too 'abstract' level, *i.e.* focuses on generic entities and relations, whereas white-box testing or code exploration tasks use more specific concepts *e.g.* implementation inheritance, dependency via type usage, or dataflow graphs.

Aspect	Score	Observations
Ease of installation	4.8	No installation problems; installation is easy
Quality of documentation	4.0	Manual needs more step-by-step examples
Scalability	4.2	Tool requires a modern computer with a fast (OpenGL accelerated) graphics card
Fact extraction	3.6	C/C++ support outside Visual Studio is limited
Visualization	4.4	Views have generic, hard to interpret, labels and annotations More customizable colormaps are required
Interaction	4.0	Selecting small elements can be hard in zoom-out mode
Effectiveness (for general comprehension)	3.9	Tool gives a good overview of a program's structure and dependencies It is a nice addition compared to classical IDE text-only views and queries
Effectiveness (for white-box testing)	2.2	Tool is too generic; needs customized wizards that should address specific questions
Usability (general)	4.0	Tool is easy to use and error-tolerant An undo function is however missing and would be very useful

Table 2: SolidSX tool evaluation in education.

Tool developer context: In the second case, 4^{th} year MSc students at the same university used SolidSX for the course Software Maintenance and Evolution (SME) (20 students per year on average during 2008-2011) to develop a visual exploration application of changing dependencies in a Subversion (SVN) repository. Hence, in contrast to the first use-case presented earlier in this section, where SolidSX was used out of the box by *end users*, in this second context, SolidSX was used as a component to *develop* a new SVA application. The repository chosen for exploration was KOffice, which has mainly C/C++ code (over 10000 files, 3500 versions, over a 8 year period) [68]. Repository access used the SharpSvn C# library [69]. From each revision, hierarchy and dependencies (inheritance, type usage, include relations, and function calls) were extracted with the lightweight CCCC analyzer [70], which uses a C/C++ parser built with the ANTLR parser generator. The resulting compound graph was stored in a SQLite database (Sec. 3.1). An exploration tool allowing the selection of the repository and revisions of interest, dependency extraction, and visualization of changing dependencies was built by the students using SolidSX as a starting point. One such tool resulting from a student project is shown in Fig. 5. Here, the HEB plot reuses the SolidSX tool; all other interface elements are custom-developed by the students for this specific project. Node colors indicate amount of code changed on a blue-to-red colormap. The student project along with manual, documentation, binaries, and C# source code is available for inspection [71]. The students were able to build atop SolidSX to develop their dependency evolution visualization without access to the tool's source code. The student tool was written in C#, while SolidSX itself is entirely written in C++ and Python. Key to this (re)use of SolidSX were the open SQLite data model (Sec. 3.1) and the message-based mechanism that allows 'driving' SolidSX from any application (Sec. 4.2.3).

5.2. Toolset Usage in Developing New Research

Apart from education, we also used SolidSX in to develop new visualization research. One such example is Image-based edge bundles (IBEB), a technique for the simplified display of large compound graphs [64]. Briefly put, IBEB adds several image-processing techniques atop of the HEB view in order to explicitly group edges into salient shaded bundles. This shows a (software) system's structure on a coarse level of detail. Like in the educational context (Sec. 5.1), the SQLite data model and messaging interface were key to easy development. However, in this case we also extended SolidSX by adding new graphical elements to the radial plot (for details, see Ersoy and Telea [64]), rather than embedding it in a larger application. Figure 6 shows some results: The left image shows a set of software dependencies visualized with the standard SolidSX tool. The right view shows the same dataset, with dependencies grouped by IBEB into shaded bundles.

5.3. Industrial Usage: Post-Mortem Assessment of a Software Project

We now describe an application where several of our tools were combined in an industrial use-case. A major automotive company developed an embedded software stack of 3.5 MLOC 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 add. 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 had *one week* to perform a post-mortem analysis using our toolset, and only the code repository as data source. For full details, we refer to Voinea and Telea [49].

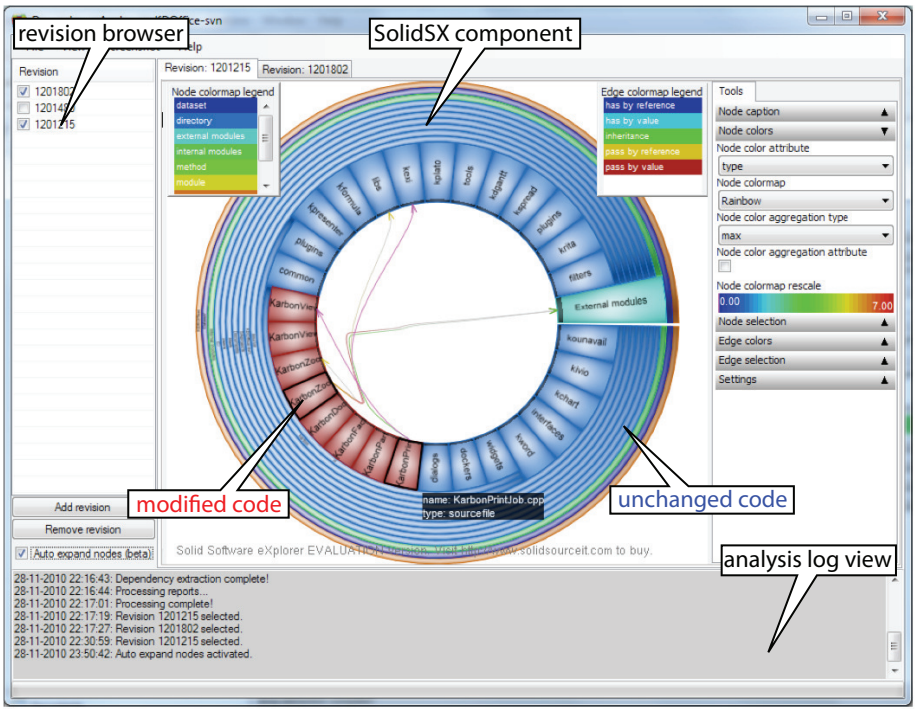

Figure 5: Subversion-repository dependency evolution browser tool interface (see Sec. 5.1).

The approach involved the classical VA steps: data acquisition, hypothesis creation, refinement, (in)validation, and presentation (Fig. 7). Change requests (CRs), commit authors, static quality metrics, and call and dependency graphs were extracted from a CM/Synergy repository into our SQLite fact database. For fact extraction, we used our C/C++ analyzer SolidFX [43]. Similar heavyweight analyzers *e.g.* Clang or Columbus could be used as well. All further visual analyses were done using SolidSX and SolidSDD. Next, we examined the distribution of CRs over project structure. Several folders with many open CRs emerged (red treemap cells in Fig. 7 (2)). These correlate well with team structure: the 'red' team owns most CRs (3). To further see if this is a problem, we viewed the CR distribution over files over time. In the table lens in Fig. 7 (4), files are shown as gray lines vertically stacked on age (oldest at bottom), and CRs are red dots. The gray area's shape shows little code size increase in the last project third (outlined in yellow), but many red dots over *all* files in this phase. These are CRs involving old files that were never closed. When seeing this image, the managers instantly recalled that the 'red' team (located in Asia) had lasting communication problems with the European teams, and added that it was a mistake to assign many CRs to this team.

We next analyzed the evolution of several quality metrics: fan-in, fan-out, number of functions and function calls, and average and total McCabe complexity. The graphs in Fig. 7 (5) show that these metrics increase little in the second project half. Hence, 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.

To find possible refactoring problems, we analyzed the project structure with SolidSX. Fig. 7 (6) shows forbidden dependencies, *i.e.* modules that interact bypassing interfaces. Fig. 7 (7) shows modules related by mutual calls, which violate the product's desired strict architectural layering. We decided to look at these specific structural problems after completing the project intake (the first day out of the one week total project duration). During this session, the two architects involved stated that their product should comply with strict interface-based communication and architectural layering. They were unsure if these desiderates were respected, and mentioned that problems in these areas would highly likely affect the ease of refactoring. The two views in Fig. 7 (6,7) suggest difficult step-by-step refactoring and also difficult unit testing. Again, these findings were in line with the impressions of the involved stakeholders.

Finally, to get more insight on the refactoring cost, we performed a code duplication analysis using SolidSDD. Finding duplicates can help in several ways. First, if a duplicated code block is refactored, then it makes sense to refactor all its

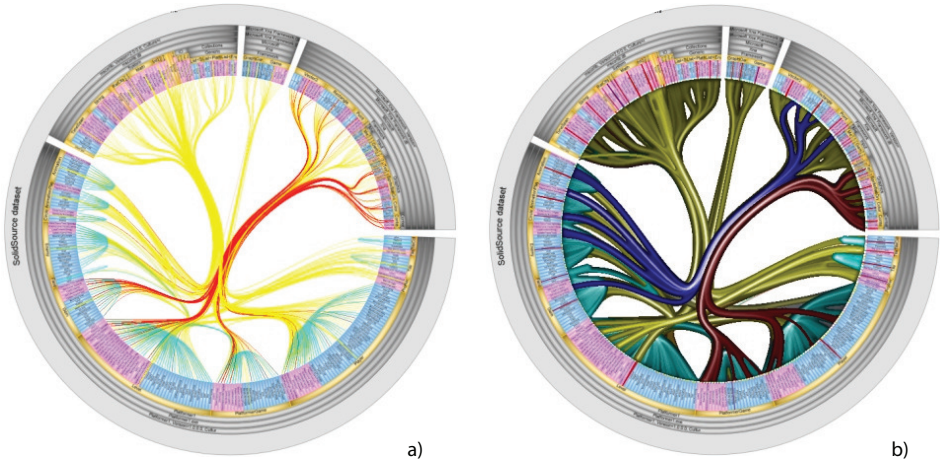

Figure 6: Left: HEB view of a compound software graph. Right: Simplified view of the same graph with the IBEB method built atop of SolidSX (Sec 5.2).

duplicates too. Secondly, code modications caused by insight found during testing and debugging should be done consistently across duplicated code. We found little cross-file duplication in this code stack, which supports the case for relative low-cost refactoring (for full details, we refer to [49]).

The consultants in this project were familiar with the used tools (SolidSX and SolidFX), but did not (need to) have access to the tools' source code. Still, they succeeded in finding satisfactory answers for the management questions in a few days and on a large code base. No modifications were done to SolidSX or SolidFX. Since SolidFX can export its dependency graphs directly into the shared SQLite database, tool communication was automatic. Besides tool installation, the only instrumentation effort required was for the extraction of evolution information (CRs, file change moments, and authors) from the CM/Synergy repository, for which we used the standard *ccm* client. This last step took approximately 4 hours of the entire project duration of one week. Post-study discussions outlined that important success factors were the easy installation of the tools, scalability, and the common (SQLite) database format that made data interchange between all involved tools very simple.

6. Discussion

Based on our SVA tool building experience, we next address several questions of interest[2]. We use herein the concept of a tool *value model* [72]: A SVA tool is *useful* if it delivers high *value* with minimal *waste* to its stakeholders, which can be developers, testers, project managers, or consultants [33, 73]. Hence, the answers to the above-mentioned questions strongly depend on the users' views on value and waste, as follows.

6.1. Should academic tools be of commercial quality?

We see two main situations. If tools are used *purely* to test new algorithms or ideas (*e.g.* the IBEB visualization in Sec. 5.2), large investments in tool infrastructure (Sec. 2) are seen as waste. If tools are used in case studies (*e.g.* Sec. 5.1) or in real-life projects (*e.g.* Sec. 5.3), then usability is key to acceptance and success [5, 74, 65]. Hence, we believe that academic tools intended for other users than their own creators should not compromise on *critical* usability, *i.e.* interactivity, scalability, and robustness. However, effort critical for the latter *adoptability* phase, *e.g.* manuals, installers, how-to's, supporting many input/output formats, rich GUIs, can be limited.

An excellent overview of the path from a research tool to a commercial product is given by Bessey *et al.* [75] for the Coverity bug-finding tool. In our experience with with SolidSX and SolidSDD, we noticed several points in line with Bessey *et al.*, as follows:

[2]The questions are from the WASDeTT 2010 call for papers (www.info.fundp.ac.be/wasdett2010)

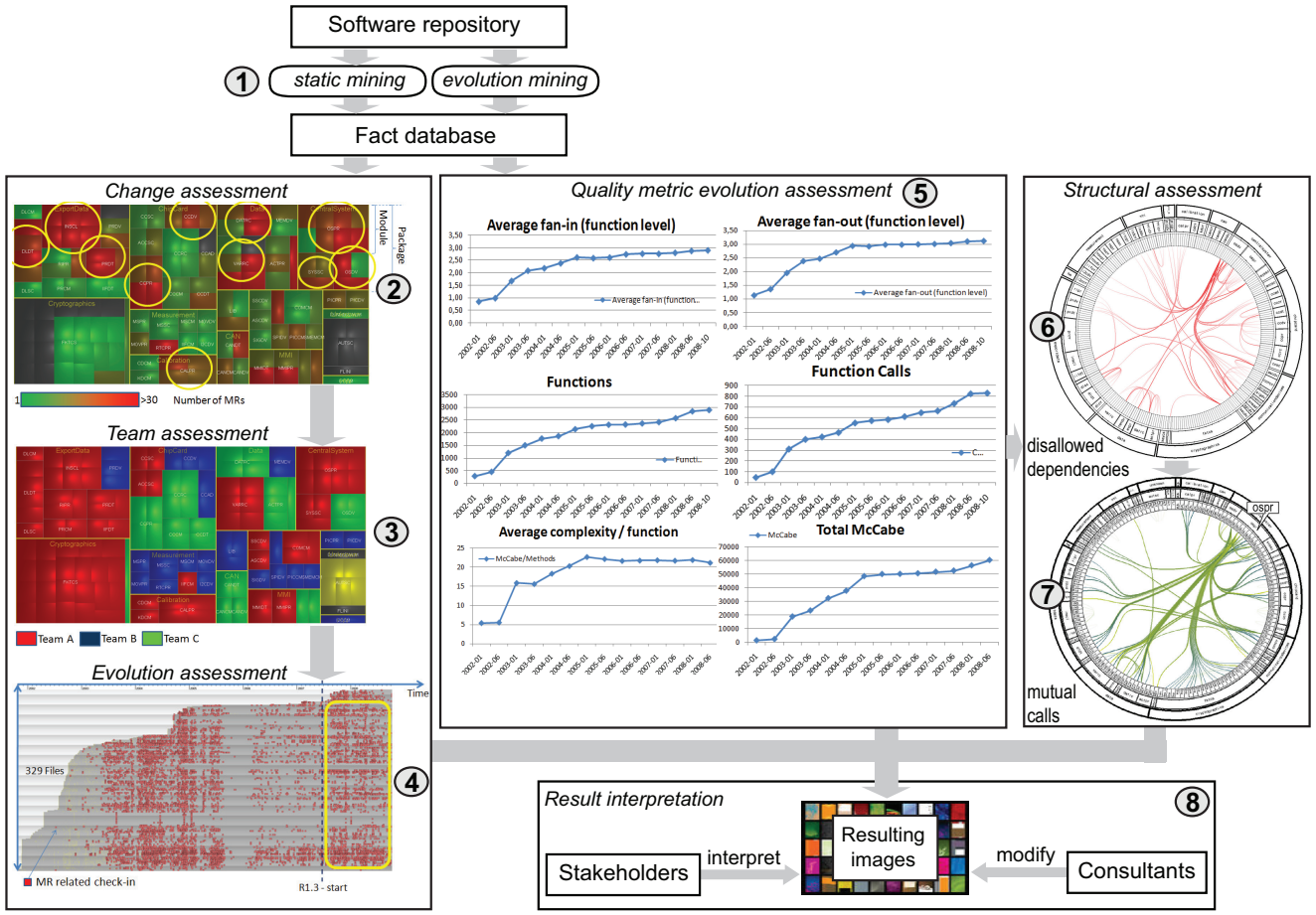

Figure 7: Data collection, hypothesis forming, and result analysis for a post-mortem software assessment (Sec. 5). Arrows show the order of the executed steps.

- industrial users tend to cluster into detractors and promoters, *e.g.* 'why would (visual) program comprehension help my job?' *vs* 'this is a great tool, no questions asked';
- early adoption in large organizations having code bases of several MLOC [42, 43] does cost significant promotion effort which cannot be easily bypassed;
- the path of least intrusion in a company's established practices *e.g.* via compiler wrapping (makefiles, build process, coding standards) is the most successful;
- countless variations of platforms, language dialects, and build systems pose hard problems to a (visual) tool acceptance;
- simple visualizations (albeit sometimes too simple) are easier accepted than subtler correlations of multiple variables which may be hard to understand.

Still, some differences exist between SVA tools and bug-checking tools such as Coverity. First, SVA is meant to provide *insight*, which for good or bad, is harder to quantify than a bug list. In other words, it is easier to measure the added value of introducing a bug-checking tool, *e.g.* as the ratio of the number of bugs found to the tool's ownership and usage costs, than to quantify how much insight a SVA tool has given. Secondly, we limited ourselves, on purpose, to support code bases where analysis is easily done with minimal configuration effort (see Sec. 4.2.1). This makes our proposal less generic than Coverity's but

also reduces tool set-up costs. Thirdly, SVA tools are mainly aimed at individual developers and/or consultants. Hence, issues such as standard compliance with a company's policy, deployment effort, and pricing are less relevant. This further lowers the hurdles for acceptance of SVA tools.

6.2. How to integrate and combine independently developed tools?

This is a highly challenging question as both the integration degree required and the tool heterogeneity vary widely in practice. For SVA tools, such as the ones described in this paper or the ones mentioned in Sec. 2, we have seen that the following patterns provide good returns on investment, in increasing order of difficulty/effort:

Dataflow: Tools communicate by reading and writing data files in standardized formats, *e.g.* SQL (tables), XML and GXL (attributed graphs) [76], and FAMIX and XMI (architecture models) [77]. This allows creating dataflow-like tool pipelines, like the excellent Cpp2Xmi UML diagram extractor using Columbus and GraphViz [78] or the SQuAVisiT toolset [79].

Shared databases: Tools communicate by reading and writing a single shared fact database which stores code, metrics, and relations. Data is typically stored as a combination of text files (code), XML (lightweight structured data), and proprietary binary formats for large datasets *e.g.* ASGs or execution traces. This model is used by Eclipse's CDT, Visual Studio's Intellisense, and our toolset. In contrast to dataflows, shared databases support the finer-grained data access needed for real-time data browsing (SolidSX, SolidSDD) or symbol queries (Eclipse, Visual Studio). If a shared messaging system is used along a shared database, like for all tools above, tools automatically synchronize their views upon data changes. This is essentially the model-view-controller pattern, where the database is the model, the visualization tools are the views, and the message listener is the controller.

Common API: Tools use a single API to access shared data and also to execute operations. Although a common API does not necessarily mean a shared tool code base, the former typically implies the latter. API examples for SVA tools are Eclipse, Visual Studio, CodeCrawler [17] and Moose [80] and, at a lower level, the Prefuse and Mondrian Infovis toolkits [81, 27]. Common APIs allow a finer grained action composition than dataflow and shared databases. However, APIs are more restrictive to use in practice, as they add non-negligible API learning costs (for tool builders) and maintenance costs (for API providers).

A thorough discussion of interoperability in the context of reverse engineering tools is provided by Kienle ([82], Sec. 3.2.2). Although our context is somewhat different (SVA tools), most, if not all, the requirements and solutions surveyed by Kienle are very similar to ours, see *e.g.* dataflow *vs* ToolBus [83], and shared databases and common APIs *vs* the CORUM model [84]. In contrast to some reverse engineering toolsets, however, we chose not to explicitly store application or task-dependent schema models. This reflects our practical observation, in line with Kienle's analysis, that schema design (and reuse) is hard in practice; and our additional observation that visualization exploration scenarios often need to change viewpoints on the data dynamically, *i.e.* in the middle of an exploration, which makes predefined schemas less effective.

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

SVA tool building is a design activity. A critical success factor is creating visual and interaction models that optimally fit the users' mental map [85]. 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 [86]. We found, in our over 10-year work in software visualization, no single case when a 3D visualization was better accepted than a 2D one. As such, we abandoned earlier work in 3D visualizations [36] 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 and visualization is mandatory for SVA acceptance [5, 73]. However, achieving this is hard. Over 50% of our toolset code (which is over 700 KLOC in total) is dedicated to efficient data manipulation. SQLite performs

well up to a few hundred thousands facts (Sec. 3). Heavyweight parsers, *e.g.* SolidFX, Clang, or Columbus create ASGs of millions of facts, roughly 10..15 facts per LOC. These are stored in a custom binary format which optimizes search speed to space ratio [43, 19]. The SQL database is used as a 'master' component which points to such special storage schemes. Using XML, although favored by several tool designs [87, 11], is simply not scalable enough for fine-grained fact databases or projects over a few hundred KLOC.

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

Quality of SVA tools depends on usability, which has different definitions depending on the tool's context. For example, research tools aimed at quickly testing new ideas should maximize API simplicity. Prefuse and Mondrian are good examples [81, 27]. In contrast, tools for software engineers in the field, or used in education, should maximize end-user effectiveness. This further implies uncluttered, scalable, and responsive displays, and tight integration for quick analysis-visualization sensemaking loops. Recent Infovis advances have massively improved the first points. However, integration remains hard, as it implies large engineering efforts which do not directly map to high-impact research results.

6.5. What is needed to build an active community of developers and users?

SVA tools rely upon techniques traditionally built in two separate communities: software analysis and information visualization. The two groups overlap via the software visualization community. The *OSS community* has invested considerable effort in SVA tool building, see *e.g.* the hundreds of plug-ins available for Eclipse, QtCreator, or KDevelop. However, most recent work in this area appears to target software analysis. Visualization is still centered around tables and node-link layouts. Although treemaps, table lenses, adjacency matrices, and HEB layouts have been proven to be more scalable and effective in the Infovis community, these techniques are still relatively unknown to OSS developers. The *commercial* community can serve as a strong catalyst: standardized open APIs of well-accepted tools, such as Visual Studio 2010's Intellisense semantic database, can give a widespread and quick impact to successful SVA tools. Unfortunately, many tool vendors provide limited tool-integration APIs, or even provide no such APIs, arguably to limit disclosure of internal architecture details [73].

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

We see the following elements present in most SVA tools we have studied:

Architecture: The dataflow and shared database models are the widest used composition patterns.

Visualization: 2D dense-pixel views *e.g.* table lenses, HEBs, treemaps, and annotated text are highly scalable, thus suitable for large static analysis datasets. Node-link layouts work well for relatively small relational datasets of less than roughly a few hundred elements, in which position and shape encode specific meaning like in (UML) diagrams (see further Sec. 6.7). Shaded cushions, originally used for treemaps [53], are simple to implement, fast, scalable, and effective for conveying structure atop of complex layouts. Also, we did not notice so far requests from users of our toolset to include other types of visualizations *e.g.* node-link layouts, adjacency matrices, parallel coordinates, or 3D plots. This does not imply that such additional visualizations are not useful for specific use-cases, but it does support the hypothesis that general structure-and-metric comprehension of large code bases at a fine grained level is well supported by the techniques currently present in out toolset.

Integration: Combining separately developed analysis and visualization tools is still an open challenge. Although a message mechanism (Sec. 4.2.3) has limitations, *e.g.* cannot shared state, it is simple and allows keeping the software stacks of the tools to integrate independent.

Static analysis granularity: Lightweight analyzers are considerably simpler to build, deploy, and (re)use, are faster, and deliver sufficient details for visualization (Sec. 4.2.1). When visualization requirements increase, as it happens with a successful tool, so do the requirements on its input analyzer. Extending static analyzers is, however, not an incremental process: For providing more features, one typically needs switching to a completely new analyzer. Using a simple fact database schema helps this switching as the analyzer and visualization are weakly coupled. Adding new analyzers to our toolset, *e.g.* for additional languages, only requires setting up a small Python script (under 30 lines) which invokes the respective analyzer and populates the standard SQLite database (Sec. 3.1) with the extracted facts. An example hereof is a Visual Basic parser, developed independently and written in Visual Basic itself, which we recently added to SolidSX [88].

6.7. How to compare or benchmark such tools?

SVA tools can be benchmarked by lab, class, or field user studies, or using them in actual IT projects. One can either compare several tools against each other [89, 58, 49] or test a tool *vs* a requirements set [32, 74]. *Technical* aspects *e.g.* speed, scalability, or analysis accuracy can be measured using *de facto* standard datasets from the ACM SOFTVIS, IEEE Vissoft, and IEEE MSR conference 'challenges', *e.g.* the Mozilla Firefox, Azureus, JUnit, or JHotDraw code bases. Measuring a tool's end-to-end *usefulness* is much harder as this is highly context specific. Still, side-by-side tool comparison can be used. For example, Figure 8 shows four SVA tools: Ispace, CodePro Analytix, SonarJ, and SolidSX. The first three are well-known in the Java community. We compared all tools for their effectiveness in supporting an industrial corrective maintenance task where participants were actual developers from the IT industry [90, 73]. Several of our design decisions, *e.g.* using dense-pixel layouts and bundled edges, are direct results of this study. Also, this study showed that integration, ease of installation, scalability, and the compact uncluttered dense-pixel layouts of SolidSX were perceived as important value factors by developers involved in program comprehension.

Other useful ways to gather qualitative feedback are 'piggybacking' the tool atop of an accepted toolchain (*e.g.* Eclipse or Visual Studio) and using 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.

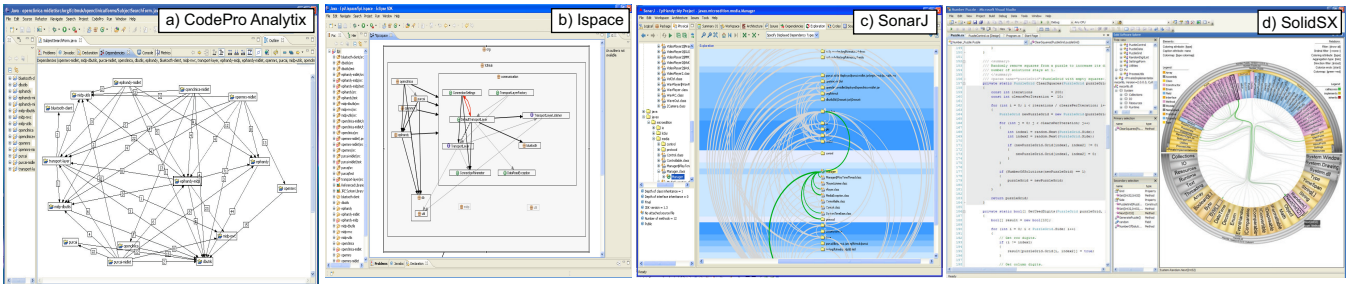

Figure 8: Four SVA tools for structure-and-association software analysis compared (see Sec. 6.7).

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

For SVA tools, our experience advocates a minimal set of proven technologies, as follows:

Graphics: OpenGL, possibly augmented with simple pixel shaders, is by far the most portable, easiest to code and deploy, and fastest, graphics solution. This experience is shared by other researchers, *e.g.* Holten [52].

Core: Scalability to millions of data items, relations, and attributes can only be achieved in programming languages like C, 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. Recently, we reimplemented almost the entire core in C# (.NET 4), using the Windows Presentation Foundation (WPF), with promising results: Speed is 80% as compared to the original C++ implementation. Additionally, WPF offers easy ways to render visualizations to offscreen canvases, which makes embedding our results into *e.g.* web pages or PDF documents trivial. However, deployment becomes slightly more complex, as users have now to install the .NET 4 framework on their platform.

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 [36]) and Smalltalk (used by [17]) are other candidates. From our experience, however, Tcl/Tk and Smalltalk require more effort for learning, deploying, and optimization. Lua [91] brings simplicity and flexibility for applications that do not require the richer features provided by Python's third-party libraries. For SVA tools, we distinguish two different uses for scripting. Generic SVA *frameworks* such as Rigi [92], Prefuse [81], Mondrian [27], or Moose [80] promote scripting to a first-class citizen: Users have to script to construct specific visualizations. This makes such frameworks very generic, but also less effective by end users not interested in, or not

having the time for, programming. In contrast, coarser-grained tools, like ours, use scripting for (internal) configuration but expose all their functionality via built-in GUI options. This inherently limits the customizability of such tools, but makes them easier and faster to use for predefined tasks.

6.9. Evolution from research prototype to product

As already mentioned, our SVA toolset has evolved from an initial set of visualization research prototypes, starting with an early Rigi-like generic visualization system (SoftVision [36], see further Tab. 1), to an actual product toolset, via an iterative design process. We next discuss several aspects of this evolution.

Design decisions: During our tool design activities, several technical design refinements took place, as follows. First, using a simple, dynamically typed database with a minimal schema (Sec. 3.1) turned out to be simpler, and easier to maintain, than specialized object-oriented data models (see *e.g.* SoftVision [36] or SolidFX [43]). While the former model uses a fixed set of four data tables (see Sec. 3.1), the latter models need hierarchies of hundreds of classes, each being specialized for a different graph type, *e.g.* ASTs or call graphs. Secondly, using plain OpenGL for rendering is faster, both as development and runtime speed, than the more complex scene graph models of toolkits such as OpenInventor [47] used in SoftVision [36], CSV [37], MetricView [40], and SolidBA [42]. In scene graphs, the content to render is first stored explicitly, and next rendered. This works best for complex imagery that is often rendered but rarely changed. In SVA tools, views change often as the user interacts with the data, *e.g.* expand or collapse nodes, so updating a scene graph is much slower than direct content rendering by OpenGL. Thirdly, our earlier tools used node-link layouts such as Sugiyama-style or spring embedders, based on the Graphviz and VCG engines [48, 25] (SoftVision [36], MetricView [40], and SolidBA [42]). However, no such engine can currently produce uncluttered layouts for graphs over a few hundred nodes. Hence, we restricted our later tools to HEB layouts, treemaps, and table lenses, which are always scalable and clutter-free. Fourthly, we experimented with data storage using binary files (SolidFX [43]) and XML files (MetricView [40], MemoView [41]). The SQLite model, used in CVSscan, CVSgrab, SolidBA, SolidSX, and SolidSDD proved to be much simpler to maintain and up to two orders of magnitude faster and more compact than XML. Finally, we kept unchanged those design elements which proved successful during our toolset evolution, such as selections and observers (introduced by SoftVision). Transition moments between these design decisions are explained further below.

Evolution phases: Our toolset evolution can be divided into four main phases:

1. *Inception:* In this phase, several SVA prototypes were created, with the main goal of exploring novel visualization and interaction technique (SoftVision, CSV, CVSscan). In this phase, a wide mix of visualization and interaction designs (*e.g.* node-link layouts, dense pixel displays, cushions, table lenses, linked views) and implementation technologies (*e.g.* C/C++ *vs* Python, OpenGL *vs* Open Inventor, SQLite *vs* plain file storage formats), were explored. The main drivers were the speed of creating new visualizations and the visualization scalability, with limited focus on reusability, genericity, or ease of deployment. The outcome of this phase was a selection of 'winner' visualization-and-interaction designs and implementation elements, *e.g.* OpenGL, cushions, table lenses, and C++, which were found to be visually and computationally scalable. In this phase, testing was done on relatively small code bases (up to a few hundreds of entities and relationships and a few thousands of lines of code), and involved mainly researchers, students, and OSS code bases.
2. *Consolidation:* In this phase, additional research prototypes were created for different exploration tasks (CVSgrab, MetricView, and MemoView). The main focus of this phase was to extend the options for SVA exploration to new types of datasets and tasks, *i.e.* entire repositories (CVSgrab), MetricView (UML diagrams and metrics) and program traces (MemoView). The design choices obtained as outcome from the previous phase were followed in this development. As they were confirmed to be effective in terms of desired scalability by the actual usage of the tools, these design choices were kept fixed until the later product refinement phase (see further below). Tool testing involved both large-scale OSS code bases (*e.g.* VTK and wxWidgets) and a few commercial code bases. For the first time, we involved IT professionals in using our tools, and collected informal feedback on usability and effectiveness, which further led to our focus on ease of deployment and configuration, and tool interoperability, addressed in the next phase. The decision to extend the tool scope to IT professionals was taken on an *ad hoc* basis, given our involvement in academy-industry joint projects. GUI

design converged to using the wxWidgets toolkit, as opposed to several variants in the inception phase (FLTK, Tcl/Tk, and Windows MFC). The datasets used in this phase were significantly larger than for the inception phase, *e.g.* file evolution information from an entire repository (CVSgrab) as opposed to a single file evolution (CVSscan), and traces of hundreds of thousands of samples (MemoView). As such, efficient storage and retrieval proved critical. SQLite emerged as a natural candidate, given the usage of SQL fact databases in other SVA tools. For the specific cases which required node-link layouts (MetricView), Graphviz and VCG emerged as best candidates in terms of ease of use, genericity, and layout quality.

3. *Initial products:* In this phase, the interactive visual metaphors and implementation techniques tried and tested in the first two phases were used as basis to create the first versions of products (SolidFX, SolidBA, and SolidSTA). This phase focused on three main additions: First, new static and repository analysis components were added, thereby extending the visualization capabilities of the earlier tools with analysis functions (SolidFX extends CSV, SolidSTA extends CVSgrab and CVSscan). Secondly, a first unified fact database model, based on SQLite, was created (see Sec. 3.1), in order to make tool interoperability possible. Thirdly, various layers were created to facilitate tool deployment, in terms of installers (NSIS-based [93]) and front-ends for automating static analysis. The technical outcome of this phase was a first version of the overall SVA toolset architecture presented in this paper.
4. *Refined products:* In this phase, a major refactoring of the visual functionality took place. Visualization components, so far scattered into different class libraries in the existing tools, were centralized in a single implementation, which led to SolidSX. The treemap and HEB components, pioneered by other research tools [53, 52], were also added from scratch. The message-based interface (Sec. 4.2.3) was developed, which allowed incorporating interactive visualization as a 'service' into analysis tools. Analysis-wise, we extended our initial scope on C/C++ with Java, C#, and .NET static analysis (SolidSX), and clone detection (SolidSDD). Deployment-wise, this phase added manuals and licensing mechanisms, which turned our toolset into a first version of true off-the-shelf end-user products. This enabled us also to conduct larger user evaluations with both IT professionals [90] and students (Sec. 5.1). Implementation-wise, we migrated from C++, OpenGL, and wxWidgets to .NET and WPF. This decision was taken due to perceived shorter development time in C#/.NET, experienced by our development team in the context of other ongoing projects, and the richer GUI options of .NET *vs* wxWidgets, including offscreen and web rendering, the latter which was required by several customers.

7. Conclusions

In this paper, we have presented our experience in developing software visual analytics (SVA) tools, starting from research prototypes and ending with a commercial toolset. During this iterative design process, our toolset has converged from a wide variety of techniques to a relatively small set of proven concepts: a single shared fact database with a simple schema, implemented in SQL, which allows tool composition by means of shared fact selections; 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 message-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 members: SolidSX for visualization of program structure, dependencies, and metrics; and SolidSDD for visualization of code clones. We outlined the added value of combining several tools in typical visual analysis scenarios by means of simple examples, academic usage in research and education, and an 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 covers extending our toolset at several levels *e.g.* lightweight zero-configuration C/C++ parsing; dynamic analysis for code coverage and execution metrics; and integration with IDEs beyond Visual Studio, *e.g.* Eclipse and KDevelop. Combining HEB layouts and annotated code text in a single scalable view to allow easy navigation from source code to structure is a second promising direction of work.

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] K. Zhang, Software visualization - From theory to practice, Kluwer Academic, 2003.
[7] P. C. Wong, J. J. Thomas, Visual Analytics, IEEE CG&A 24 (5) (2004) 20–21.
[8] J. J. Thomas, K. A. Cook, Illuminating the Path: The Research and Development Agenda for Visual Analytics, National Visualization and Analytics Center, 2005.
[9] LLVM Team, Clang C/C++ analyzer home page, `clang.llvm.org`, 2010.
[10] D. Quinlan, ROSE: Compiler support for object-oriented frameworks, in: Proc. Conf. Parallel Compilers (CPC), 81–90, see also `www.rosecompiler.org`, 2000.
[11] Y. Lin, R. C. Holt, A. J. Malton, Completeness of a Fact Extractor, in: Proc. WCRE, IEEE, 196–204, 2003.
[12] R. Ferenc, A. Beszédes, M. Tarkiainen, T. Gyimóthy, Columbus – Reverse Engineering Tool and Schema for C++, in: Proc. ICSM, IEEE, 172–181, 2002.
[13] Eclipse project, Eclipse CDT framework for C/C++, `www.eclipse.org/cdt`, 2010.
[14] S. McPeak, The Elsa C++ static analyzer, `scottmcpeak.com/elkhound/sources/elsa`, 2010.
[15] A. Ludwig, Recoder Java analyzer, `recoder.sourceforge.net`, 2010.
[16] M. van den Brand, J. Heering, P. Klint, P. Olivier, Compiling language definitions: the ASF+SDF compiler, ACM TOPLAS 24 (4) (2002) 334–368.
[17] M. Lanza, *CodeCrawler* - Polymetric Views in Action, in: Proc. ASE, 394–395, 2004.
[18] SciTools, Inc., Understand for C/C++, `www.scitools.com`, 2010.
[19] 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.
[20] T. Mens, S. Demeyer, Software Evolution, Springer, 2008.
[21] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice, Springer, 2006.
[22] N. Fenton, S. Pfleeger, Software Metrics: A Rigorous and Pracical Approach, Chapman & Hall, 1998.
[23] S. Diehl, Software Visualization Visualizing the Structure, Behaviour, and Evolution of Software, Springer, 2007.
[24] H. Kienle, H. A. Müller, Rigi—An environment for software reverse engineering, exploration, visualization, and redocumentation, Science of Computer Programming 75 (4) (2010) 247–263.
[25] G. Sander, Graph Layout through the VCG Tool, in: Proc. Graph Drawing, Springer, 194–205, see also `rw4.cs.uni-sb.de/~sander/`, 1994.
[26] AbsInt Inc., aiSee Graph Layout Software, `www.aisee.com`, 2010.
[27] A. Lienhardt, A. Kuhn, O. Greevy, Rapid Prototyping of Visualizations using Mondrian, in: Proc. IEEE Vissoft, 67–70, 2007.
[28] A. Marcus, L. Fend, J. I. Maletic, 3D Representations for Software Visualization, in: Proc. ACM SoftVis, 27–36, 2003.
[29] R. Wettel, M. Lanza, Program Comprehension through Software Habitability, in: Proc. ICPC, IEEE, 231–240, 2007.
[30] R. Wettel, M. Lanza, Visual Exploration of Large-Scale System Evolution, in: Proc. WCRE, IEEE, 219–228, 2008.
[31] S. Eick, S. Steffen, E. Sumner, Seesoft - A Tool for Visualizing Line Oriented Software Statistics, IEEE TSE 18 (11) (1992) 957–968.
[32] H. Kienle, H. A. Müller, Requirements of Software Visualization Tools: A Literature Survey, in: Proc. IEEE Vissoft, 92–100, 2007.
[33] A. Telea, A. Voinea, H. Sassenburg, Visual Tools for Software Architecture Understanding: A Stakeholder Perspective, IEEE Software 27 (6) (2010) 46–53.
[34] SolidSource BV, SolidSX, SolidSDD, SolidSTA, and SolidFX tool distributions, `www.solidsourceit.com`, 2010.
[35] SVCG, Scientific Visualization and Computer Graphics Group, Univ. of Groningen, Software Visualization and Analysis, `www.cs.rug.nl/svcg/SoftVis`, 2010.
[36] A. Telea, A. Maccari, C. Riva, An Open Toolkit for Prototyping Reverse Engineering Visualizations, in: Proc. Data Visualization (IEEE VisSym), IEEE, 67–75, 2002.
[37] 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.
[38] L. Voinea, A. Telea, J. J. van Wijk, CVSscan: visualization of code evolution, in: Proc. ACM SOFTVIS, 47–56, 2005.
[39] L. Voinea, A. Telea, Visual Querying and Analysis of Large Software Repositories, Empirical Software Engineering 14 (3) (2009) 316–340.
[40] M. Termeer, C. Lange, A. Telea, M. Chaudron, Visual exploration of combined architectural and metric information, in: Proc. IEEE Vissoft, 21–26, 2005.
[41] S. Moreta, A. Telea, Multiscale Visualization of Dynamic Software Logs, in: Proc. EuroVis, 11–18, 2007.
[42] A. Telea, L. Voinea, Visual Software Analytics for the Build Optimization of Large-scale Software Systems, Computational Statistics 26 (4) (2011) 635–654.
[43] A. Telea, L. Voinea, An Interactive Reverse-Engineering Environment for Large-Scale C++ Code, in: Proc. ACM SOFTVIS, 67–76, 2008.
[44] SQLite Team, The SQLite Database, `www.sqlite.org`, 2011.
[45] I. Kaplan, Implementing Graph Pattern Queries on a Relational Database, in: Technical Report LLNL-TR-400310, Lawrence Livermore National Laboratory, USA, 2008.
[46] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, D. Wilkins, A Comparison of a Graph Database and a Relational Database: A data provenance perspective, in: Proc. ACM SE, 68–80, 2010.
[47] VSG Inc., OpenInventor Toolkit, `vsg3d.com/open-inventor/sdk`, 2011.
[48] AT&T, The GraphViz Package, `www.graphviz.org`, 2010.
[49] L. Voinea, A. Telea, Case Study: Visual Analytics in Software Product Assessments, in: Proc. IEEE Vissoft, 65–72, 2009.
[50] 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.
[51] A. Telea, Combining extended table lens and treemap techniques for visualizing tabular data, in: Proc. EuroVis, 51–58, 2006.
[52] D. Holten, Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data, in: Proc. IEEE InfoVis, 741–748, 2006.
[53] B. Shneiderman, Treemaps for space-constrained visualization of hierarchies, www.cs.umd.edu/hcil/treemap-history, 2010.
[54] A. Telea, Software Quality Assurance and Testing (SQAT) Course Assignment, univ. of Groningen, the Netherlands, www.cs.rug.nl/~alext/SQAT/Assignment, 2010.
[55] VTK Team, The Visualization Toolkit (VTK) home page, www.vtk.org, 2010.
[56] Redgate Inc., Reflector .NET API, www.red-gate.com/products/reflector, 2010.
[57] Bell Labs, CScope, cscope.sourceforge.net, 2007.
[58] M. Sensalire, P. Ogao, A. Telea, Evaluation of Software Visualization Tools: Lessons Learned, in: Proc. IEEE Vissoft, 156–164, 2009.
[59] E. Juergens, F. Deissenboeck, B. Hummel, CloneDetective - A Workbench for Clone Detection Research, in: Proc. ICSE, IEEE, 98–107, 2010.
[60] Z. Jiang, A. Hassan, R. C. Holt, Visualizing Clone Cohesion and Coupling, in: Proc. APSEC, 130–137, 2006.
[61] T. Kamiya, CCfinder clone detector home page, www.ccfinder.net, 2010.
[62] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large-Scale Source Code, IEEE TSE 28 (7) (2002) 654–670.
[63] H. Hoogendorp, O. Ersoy, D. Reniers, A. Telea, Extraction and Visualization of Call Dependencies for Large C/C++ Code Bases: A Comparative Study, in: Proc. ACM VISSOFT, 137–145, 2009.
[64] A. Telea, O. Ersoy, Image-based Edge Bundles: Simplified Visualization of Large Graphs, Comp. Graph. Forum 29 (3) (2010) 65–74.
[65] A. Telea, L. Voinea, A Tool for Optimizing the Build Performance of Large Software Code Bases, in: Proc. IEEE CSMR, 153–156, 2008.
[66] A. Telea, Image Inpainting Tool Source Code, www.cs.rug.nl/~alext/SQAT/Software, 2010.
[67] A. Telea, An Image Inpainting Technique Based on the Fast Marching Method, J. of Graphics Tools 9 (1) (2004) 23–34.
[68] KOffice Team, KOffice Software Repository, www.koffice.org, 2010.
[69] S. Team, SharpSVN C# Library, sharpsvn.open.collab.net, 2010.
[70] T. Littlefair, C and C++ Code Counter, sourceforge.net/projects/cccc, 2007.
[71] M. Ettema, E. Vast, Dependency Evolution Analyzer, www.cs.rug.nl/svcg/SoftVis/DepEvol, 2010.
[72] M. Poppendieck, T. Poppendieck, Lean Software Development: An Agile Toolkit for Software Development Managers, Addison-Wesley, 2006.
[73] A. Telea, L. Voinea, O. Ersoy, Visual Analytics in Software Maintenance: Challenges and Opportunities, in: Proc. EuroVAST, Eurographics, 65–70, 2010.
[74] M. Sensalire, P. Ogao, A. Telea, Classifying desirable features of software visualization tools for corrective maintenance, in: Proc. ACM SOFTVIS, 87–90, 2008.
[75] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. H. Gros, A. Camsky, S. McPeak, D. Engler, A few billion of lones of code later: Using static analysis to find bugs in the real world, Comm. of the ACM 53 (2) (2010) 66–75.
[76] R. Holt, A. Winter, A. Schurr, GXL: Towards a standard Exchange Format, in: Proc. WCRE, 162–171, 2000.
[77] S. Tichelaar, S. Ducasse, S. Demeyer, FAMIX and XMI, in: Proc. WCRE, 296–300, 2000.
[78] 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.
[79] M. van den Brand, S. Roubtsov, A. Serebrenik, SQuAVisiT: A Flexible Tool for Visual Software Analytics, in: Proc. CSMR, 331–332, 2009.
[80] O. Nierstrasz, S. Ducasse, T. Gîrba, The Story of Moose: an Agile Reengineering Environment, in: Proc. ACM ESEC/FSE, 1–10, 2005.
[81] Prefuse, The Prefuse Information Visualization Toolkit, prefuse.org, 2010.
[82] H. Kienle, Building Reverse Engineering Tools with Software Components, Ph.D. thesis, Univ. of Victoria, Canada, 2006.
[83] H. de Jong, P. Klint, ToolBus: The next generation, in: F. de Boer, M. Bonsangue, S. Graf, W. de Roever (Eds.), Formal Methods for Components and Objects, Springer LNCS, 220–241, 2003.
[84] R. Kazman, S. Woods, J. Carriere, Requirements for integrating software architecture and reengineering models: CORUM II, in: Proc. WCRE, 154–163, 1998.
[85] J. J. van Wijk, T. Isenberg, J. Roerdink, A. Telea, M. Westenberg, Visual Analytics Evaluation, in: Mastering the Information Age: Solving Problems with Visual Analytics, chap. 8, Eurographics, 131–143, 2010.
[86] A. Teyseyre, M. Campo, An Overview of 3D Software Visualization, IEEE TVCG 15 (1) (2009) 87–105.
[87] Gccxml Team, The Gccxml C++ parser, www.gccxml.org, 2011.
[88] L. Kwakman, Automatically Reducing Code Duplication, MSc thesis, Univ. of Groningen, the Netherlands, www.cs.rug.nl/~alext/PAPERS/MSc/kwakman10.docx, 2010.
[89] G. Ellis, A. Dix, An explorative analysis of user evaluation studies in information visualisation, in: Proc. AVI Workshop on Beyond Time and Errors: Novel Evaluation methods for information visualization, 2006.
[90] 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.
[91] Lua Team, The Lua Programming Language, www.lua.org, 2011.
[92] S. Tilley, K. Wong, M. Storey, H. Müller, Programmable Reverse Engineering, Intl. J. Software Engineering and Knowledge Engineering 4 (4) (1994) 501–520.
[93] NSIS Team, NSIS Installer, nsis.sourceforge.net, 2012.