Visual exploration of program structure, dependencies and metrics with SolidSX

Dennie Reniers
SolidSource BV, the Netherlands
Email: dennie.reniers@solidsourceit.com

Lucian Voinea
SolidSource BV, the Netherlands
Email: lucian.voinea@solidsourceit.com

Alexandru Telea
University of Groningen, the Netherlands
Email: a.c.telea@rug.nl

***Abstract*—We present SolidSX, an visual analysis tool for code structure, dependencies, and metrics. Our tool facilitates the understanding of large program code bases by simplifying the entire pipeline from data acquisition up to visualization and interactive querying. Secondly, SolidSX is an easy to use, scalable, and configurable visualization component for compound attributed graphs extracted by third-party tools, easy to integrate by developers in their own applications. We detail the architecture and functions of SolidSX, present examples for its two use-cases, and outline insights collected from tool usage in academia and industry.**

I. Introduction

Program comprehension reportedly costs as much as 40% of the software lifecycle. In the past decade, tens of visualization tools for program comprehension have emerged. Many such tools share the same conceptual data model: From raw data *e.g.* code files, a *compound attributed graph* (CAG) is extracted. Nodes encode program entities, *e.g.* folders, files, classes, and methods; containment edges encode the program's hierarchy; association edges encode entity dependencies, *e.g.* uses, inherits, imports, or calls; and key-value attributes on nodes and edges encode *e.g.* names, metrics, or annotations.

However, such tools have limited impact in the IT industry. Key reasons are limited visual scalability, long learning curves, and poor integration with development toolchains [20], [2], [10]. Different attempts to solve these problems exist. Dense graph layouts target scalability, *e.g.* treemaps (CodeCity [31], EvoSpaces [12]); directed-tree and SHriMP layouts (Mondrian [27], CodeCrawler [3]); adjacency matrices (MatrixZoom [1], NDepend, Lattix); and multidimensional scaling (Codemap [11]). End-user tools have simple installers, minimal configuration and programming, and rely heavily on predefined queries (NDepend, Lattix LDV, MatrixZoom [1]). Developer and research tools focus on genericity and customizability *e.g.* via scripting, but have longer learning curves (Rigi [28], Mondrian). Plug-ins for IDEs (Eclipse, KDevelop, Visual Studio) and interchange input formats (FAMIX, GXL, XMI) address toolchain integration.

In this paper, we present SolidSX, our quest to designing a simple to use, yet generic and flexible, software visualization tool for CAGs. Along the model of Maletic *et al.* [17], SolidSX supports the *task* of visual exploration of large CAGs for program comprehension; its *audience* includes end-users who want to use the tool on their code in a few minutes, and developers, who want to (re)use the tool's visualizations to build custom applications; the visualization *target* is a generic CAG with any number and/or type of hierarchies, associations, and attributes; the visualization *medium* is the standard PC display; and the *representation* uses three linked views based on treemaps, hierarchical edge bundling, and table lenses. We next present the tool's design decisions (Sec. II), the tool's three views (Sec. III), and sample end-user and developer usage (Sec. IV). Section V concludes the paper.

II. Architecture

SolidSX has a layered dataflow architecture (Fig. 1). The input CAG comes in XML format or is extracted by static analysis plug-ins in the tool's own front-end (Sec. II-B). The core layer implements the CAG data storage and querying (Sec. II-A) and views (Sec. III). The interface layer offers view management for embedding in third-party tools (Sec. II-C).

A. Core layer

We store our CAG data in a SQLite database. Besides nodes and edges, called *facts*, we store two other items: selections and attributes. *Selections* are sets of facts or other selections, created interactively (click and select visible facts), by scripts, or stored in the input data. Selections group semantically related elements in dynamic, user-driven, task-specific, ways. They have unique names by which they are referred in views or queries. *Attributes* are named numerical, ordinal, categorical, or text values. Facts can have any number of attributes with different names (keys). Attributes are computed by queries or filters (*e.g.* complexity, fan-in, fan-out, cohesion, coupling) or interactively set by users (*e.g.*, annotate certain classes as being 'unsafe').

SQLite can be inefficient for multiple-table joins [8], [30]. We addressed this by the following schema (Fig. 2). Facts have unique primary-key IDs. A *hierarchy* table stores one containment edge per row, listed as (parent, child) node IDs. An *association* table stores one association edge per row, listed as (from, to) node IDs. A *node attribute* table stores all attributes $a_1, \ldots, a_n$ of a node per row as n columns named by the attribute keys. An *edge attribute* table does the same for edge attributes. Edge types, *e.g.* calls, includes, are stored as edge-type attributes. For each selection, two *selection* tables store its node and edge IDs. Selection-specific fact attributes are added as extra columns. This schema captures any CAG *e.g.* class hierarchies, call graphs, or clone relations. Fig. 2 bottom shows an example. The hierarchy has a file *main.cc* with the *main()* and *run(Foo)* functions, and a class *Foo* with a method *load()*. Associations capture call; define; and 'uses type' relations (*run(Foo)* uses the type *Foo*), encoded as edge 'type' attributes. Nodes have name and lines-of-code (LOC) attributes.

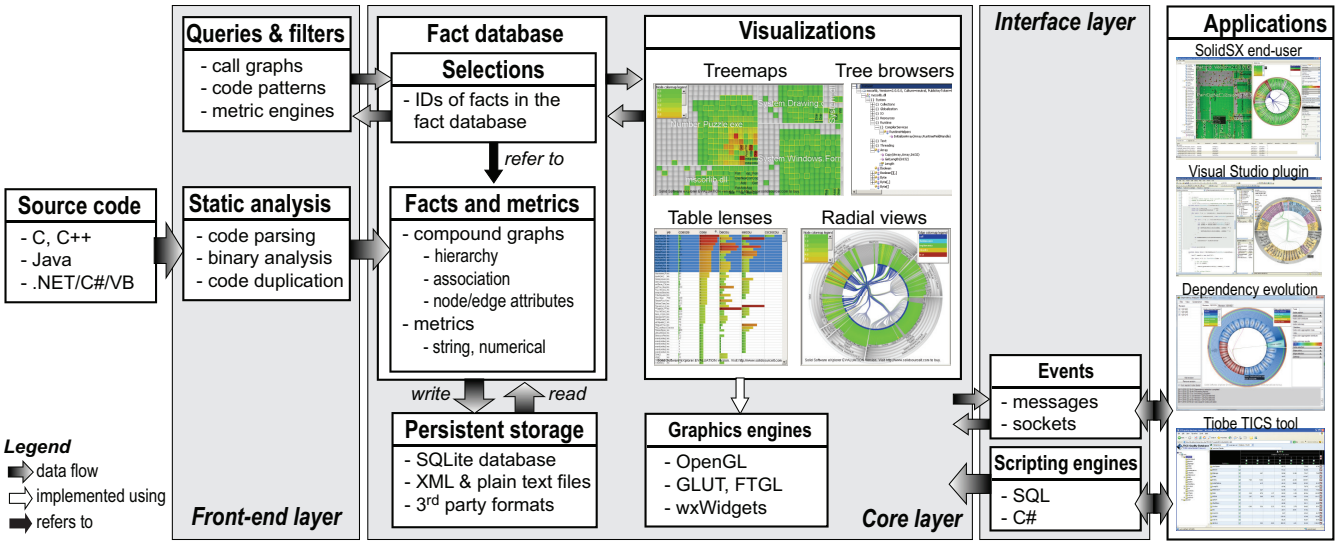

Fig. 1. SolidSX tool architecture layers and dataflows (see Sec. II)

Two selections model *main()*'s call graph (red) and *run(Foo)*'s requires graph (green)[1].

Analyses and visualizations are weakly typed: They all read, and optionally create, selections (Fig. 1). This allows composing visual analyses using selections as 'glue', statically or at runtime, with virtually no configuration costs. Components decide internally how they execute their task on a given input selection.

Computing selections, annotations, or layout properties on-the-fly imply creating, editing, and deleting hundreds of selection tables or attribute columns in a typical scenario. Separate tables for each selection optimizes speed and memory load; missing values are naturally handled by SQLite; so our schema scales well to databases of hundreds of thousands of facts with tens of attributes per fact [7]. Several hierarchies can be added as multiple hierarchy tables. Simple queries and metrics can be directly implemented in SQL. Traversing a graph for structure queries (*e.g.* connected components or reachability) or rendering is efficient, by iterating over the node and edge tables. For example, rendering the CAG in Fig. 1 top (4000 nodes and 15000 edges), takes under 0.05 seconds on a commodity PC.

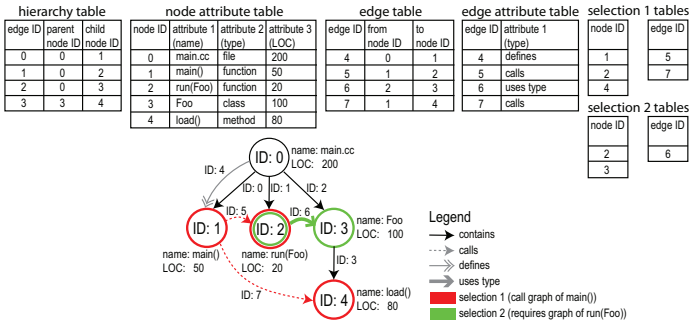

Fig. 2. Database (top) for a simple CAG (bottom) with the call graph of *main()* and the 'requires' graph of *run(Foo)* (Sec. II-A)

B. Static Analysis Front-end

SolidSX provides several parsers: Recoder (Java) [16], Reflector (.NET/C#) [19], and Microsoft's *bscsdk* parser (Visual C++ *.bsc* symbol files). These provide CAGs with folder-file-class-method and namespace-class-method hierarchies, dependencies (calls, symbol usage, inheritance, and package/header inclusion), and basic metrics (LOC, comments, complexity, fan-in, fan-out, and symbol location). Recoder, Reflector, and *bscsdk* are lightweight, robust, and fast (roughly 100KLOC/second), perfect for on-the-fly structure-and-dependency visualization. For .NET/VB/C#, Java, and Visual C++, static analysis is *fully* automated: Users pass a code root directory and, for Java, optional classpaths. C/C++ analysis beyond Visual C++ uses CAGs created by the external SolidFX C/C++ analyzer [24]. SolidFX scales to millions of LOC, covers gcc, C89/99, and ANSI C++, and handles incorrect and incomplete code. Integration with other heavyweight C++ analyzers *e.g.* Columbus [5] or Clang [15], although not yet done, is easy – we only need to convert the analyzer's output to SolidSX's XML input. Using *lightweight* C++ analyzers *e.g.* CPPX [13], gccxml, and MC++ is ineffective as these deliver incorrect data due to simplified preprocessing and name lookup. The C/C++ parsers of Eclipse CDT, KDevelop, QtCreator, and Cscope are slightly better in correctness, but are not designed as reusable components.

C. Toolchain Integration

SolidSX's visualizations can be added to existing analysis tools. Rather than offering fine-grained visualization APIs like Rigi, Mondrian, or CodeCrawler, we took a coarse-grained, black-box, approach. SolidSX listens for asynchronous Windows command messages, *e.g.* load a dataset, zoom on some subset, change view parameters, and also sends user interaction events, *e.g.* user has selected a fact, as messages. Hence, SolidSX can be embedded in any third-party tool via thin wrappers which read, emit, and process such messages. No access to SolidSX's source code is needed. For example, we integrated SolidSX in Visual Studio by writing a plug-in of around 200 LOC which translates between the IDE and SolidSX events (Fig. 3 bottom). Selecting and browsing code in the two tools is now in sync. The open SQLite format further simplifies data-level integration. Adding SolidSX to Eclipse, KDevelop,

[1]We highly recommend viewing this paper in full color

and QtCreator is under way, once we finalize the importers from these IDEs' fact databases into our SQL database.

III. Views

We limited SolidSX to a few visualizations (Fig. 3 top): *Table lenses* draw cells as pixel bars scaled and colored by attribute values [18]. *Hierarchically bundled edges* (HEBs) show CAGs by bundling association edges along hierarchy edges [6]. *Squarified cushion treemaps* compactly show structure and metrics for tens of thousands of facts [21].

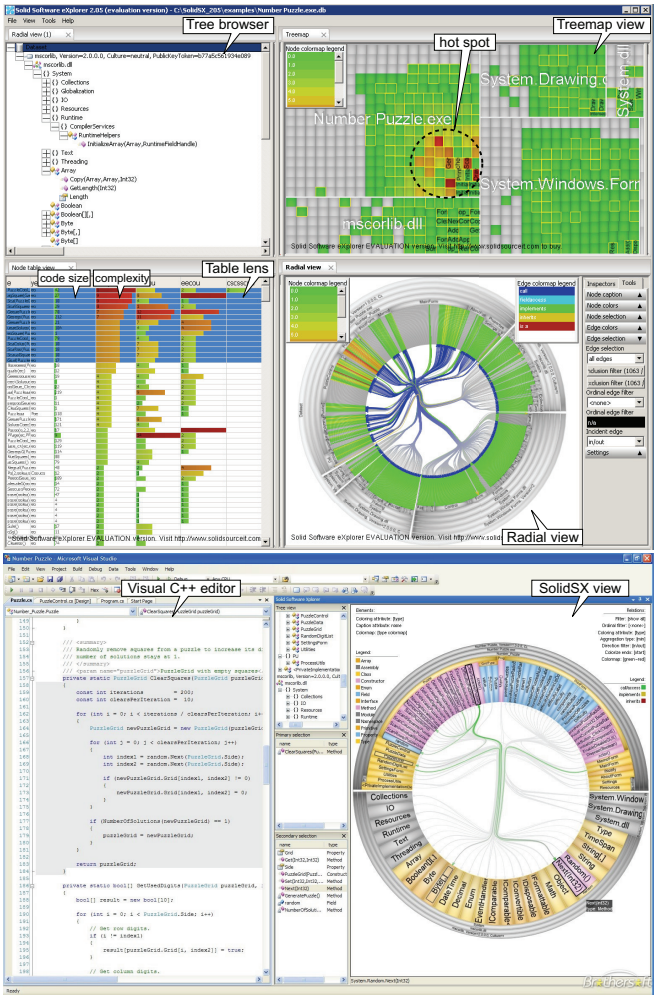

Fig. 3. Top: SolidSX views (tree browser, treemap, table lens, radial); Bottom: Visual Studio integration of SolidSX (see Sec. II-C)

Each view has an user-specified *input selection* in the fact database. If views share an input selection, changes to this selection automatically show up in the views (the linked view concept). Interaction behaves similarly: selecting items in any view, *e.g.* by mouse picking, updates a global *user selection*. Shared selections allow to easily create complex analyses of structure, dependencies, and attributes along different viewpoints. Views only receive selection *names* as input and pull their data on demand, so large datasets are efficiently handled by reference. Additionally, SolidSX provides classical tree browsers, legends, annotations, details-on-demand at the mouse, and attribute-based searches. Predefined colormaps are suggested based on attribute types *i.e.* ordinal, categorical, or numerical. Views have carefully designed *presets* so they can be used with no extra customization.

We extended the original HEB layout from [6] in several ways. Luminance cushion textures on nodes emphasize the hierarchical structure. When nodes are collapsed or expanded, the layout is smoothly animated between the initial and final views, which helps maintaining the so-called mental map (see the actual tool or tool videos at [22]). Multiple edges between collapsed nodes are visually rendered as a single edge. If edge color mapping is on, this edge shows the aggregated value (min, max, or average) of the collapsed edges' attributes, as specified by user preferences. Adjacent nodes smaller than a few pixels are replaced by gray, untextured, bars. This tells that the view cannot fully show its input and also keeps a high frame-rate regardless of dataset size, since the amount of nodes drawn never exceeds the view size divided by the minimal node size. We use the same technique for the table lens and treemap views.

Treemap views can be customized. First, users can select which levels they want to see out of the total number of hierarchy levels. Skipped levels are removed on-the-fly when laying out the treemap, which is very fast. For each selected level, one can choose a different layout: slice-and-dice, squarified, strip [21], or sorted, and also specify attribute names whose values to use for node color, size, label, and order. Secondly, we propose an adaptation of the squarified layout where we keep sibling node sizes constant but lay them out sorted according to an attribute value. This arranges siblings in their parent cell from the top-left to the bottom-right corner. Mapping a second attribute to *e.g.* color allows one to quickly see correlations (or lack thereof) between two attributes of a hierarchy.

Figure 3 (top) shows SolidSX's views on a C# system (45 KLOC). The HEB view shows function calls and system structure. Calls go from blue to gray. Node colors show McCabe's metric on a green-to-red colormap, thereby enabling structure-complexity correlations. We see *e.g.* that the most complex functions (warm colors) are in the classes located top-left in the HEB view. The table lens view shows several function-level code metrics, sorted on decreasing complexity, *i.e.* how different metrics correlate. In Fig. 3, we see that complexity is not correlated to code size. Alternatively, one can select *e.g.* the most complex or largest functions in the table lens and see where they appear in the HEB or treemap views. The treemap view shows a flattened system hierarchy (only module and function levels are selected); functions are colored on complexity and ordered from the top-left to the bottom-right corner of their parent cells on code size using the sorted layout. The 'hot spot' in the figure shows a module that has a certain amount of size-complexity correlation, but not a perfect one: A perfect correlation would yield a continuous red-to-green color gradient along the module cell's diagonal. Building the *entire* scenario, static analysis included, took about 2 minutes and under 20 mouse clicks.

IV. Tool Availability and Usage

SolidSX is written in C++ with wxWidgets (GUI) and OpenGL 1.1 (rendering). A Windows-based installer, manuals,

videos, and sample data are available [22]. SolidSX's treemap, table lens, and radial views were used in earlier code quality assessments [7], [26], [25]. Their combination in one tool and black-box reuse mechanism shown here are new. SolidSX has been used for three years by over 100 students in lectures on software quality assurance, testing, and maintenance [23]. Lecture feedback shows that SolidSX needs around 15 minutes to install and learn. SolidSX was also used to visualize evolving dependencies in a Subversion (SVN) repository. Data mining used the SharpSvn C# library. Inheritance, type usage, and include relations were extracted from each version with the CCCC analyzer [14] into SolidSX's SQL database (Sec. II-A). This tool reuses SolidSX's HEB view to show dependencies in a user-selected version, colored by type or evolution status (added in the current revision, deleted in the next revision, or persistent between two revisions). Figure 4 shows a snapshot from this tool for the KOffice repository (over 10000 files, 3500 versions, over a 8 year period) [9]. The developers who created this evolution visualization tool were in no way familiar with SolidSX, did not have access to its source code, and used a different programming language (C#) than in SolidSX (C++). The keys to reuse were SolidSX's open SQL data model (Sec. II-A) and the message-based mechanism that allows 'driving' SolidSX from any application via Windows messages (Sec. II-C). Source code and manuals of our evolution visualization tool are available [4].

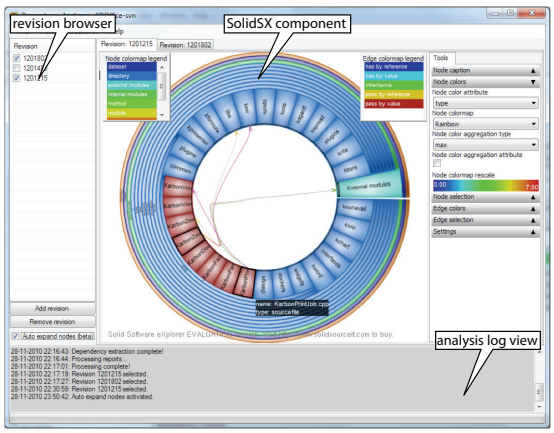

Fig. 4. Subversion dependency evolution browser with SolidSX reuse

V. Conclusions

We have presented SolidSX, an integrated tool for visualizing large software compound attributed graphs. SolidSX simplifies the task of end-users interested in visualizing such data by several design decisions: choice of highly scalable, visually stable, layouts (table lens, treemaps, and radial plots); tight integration with automated static analyzers for Java, .NET, and Visual C++; and a simple, fixed-schema, fast SQL database for data storage and querying. An event-based mechanism allows black-box tool reuse with no source-code knowledge for custom tool construction and toolchain integration. Several applications in research, consulting, and education show that SolidSX is an effective, efficient, and simple solution for visualizing the structure, dependencies, and attributes of large software systems.

Ongoing work covers extending SolidSX with UML diagram views, evolving compound graphs, integration in the TICS coding standard assessment framework [29]. and the provision of all views as a web service infrastructure for easier deployment in client-server environments.

References

[1] J. Abello and F. van Ham. MatrixZoom: A visual interface to semi-external graphs. In *Proc. InfoVis*, pages 183–190, 2005.

[2] S. Charters, N. Thomas, and M. Munro. The end of the line for Software Visualisation? In *Proc. IEEE Vissoft*, pages 27–35, 2003.

[3] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE TSE*, 31(1):75–90, 2005.

[4] M. Ettema and E. Vast. Dependency evolution analyzer, 2010. www.cs.rug.nl/svcg/SoftVis/DepEvol.

[5] R. Ferenc, A. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus – reverse engineering tool and schema for C++. In *Proc. ICSM*, pages 172–181. IEEE, 2002.

[6] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. In *Proc. IEEE InfoVis*, pages 741–748, 2006.

[7] H. Hoogendorp, O. Ersoy, D. Reniers, and A. Telea. Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study. In *Proc. ACM Vissoft*, pages 137–145, 2009.

[8] I. Kaplan. Implementing graph pattern queries on a relational database. In *Tech. Rep. LLNL-TR-400310*. Lawrence Livermore Natl. Lab., 2008.

[9] KOffice Team. KOffice software repository, 2011. www.koffice.org.

[10] R. Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *J. Soft. Maint. and Evol.*, 15(2):87–109, 2003.

[11] A. Kuhn, D. Erni, P. Loretan, and O. Nierstrasz. Software cartography: Thematic software visualization with consistent layout. *JSME*, 22(3):191–210, 2010.

[12] M. Lanza, H. Gall, and P. Dugerdil. EvoSpaces: Multi-dimensional navigation spaces for software evolution. In *Proc. CSMR*, pages 293–296, 2009.

[13] Y. Lin, R. C. Holt, and A. J. Malton. Completeness of a fact extractor. In *Proc. WCRE*, pages 196–204. IEEE, 2003.

[14] T. Littlefair. C/C++ code counter, 2007. sourceforge.net/projects/cccc.

[15] LLVM Team. Clang C/C++ analyzer home page, 2011. clang.llvm.org.

[16] A. Ludwig. Recoder java analyzer, 2010. recoder.sourceforge.net.

[17] J. Maletic, A. Marcus, and J. Collard. Atask oriented view of software visualization. In *Proc. Vissoft*, pages 57–65, 2002.

[18] R. Rao and S. Card. The table lens: Merging graphical and symbolic representations in an interactive focus+context visualization for tabular information. In *Proc. CHI*, pages 222–230. ACM, 1994.

[19] Redgate Inc. Reflector .NET API, 2011. www.red-gate.com/products.

[20] S. Reiss. The paradox of software visualization. In *Proc. IEEE Vissoft*, pages 59–63, 2005.

[21] B. Shneiderman. Treemaps for space-constrained visualization of hierarchies, 2011. www.cs.umd.edu/hcil/treemap-history.

[22] SolidSource. SolidSX Software eXplorer, 2011. www.solidsourceit.com.

[23] A. Telea. Software quality assurance and testing (sqat) course assignment, 2010. Univ. of Groningen, the Netherlands, www.cs.rug.nl/∼alext/SQAT/Assignment.

[24] A. Telea and L. Voinea. An interactive reverse-engineering environment for large-scale C++ code. In *Proc. ACM SOFTVIS*, pages 67–76, 2008.

[25] A. Telea and L. Voinea. A tool for optimizing the build performance of large software code bases. In *Proc. IEEE CSMR*, pages 153–156, 2008.

[26] A. Telea and L. Voinea. Visual software analytics for the build optimization of large-scale software systems. *Comp. Stat.*, 26(3), 2011.

[27] M. Theus and S. Urbanek. *Interactive Graphics for Data Analysis: Principles and Examples (Computer Science and Data Analysis)*. CRC Press, 2008.

[28] S. Tilley, K. Wong, M. Storey, and H. Müller. Programmable reverse engineering. *Intl. J. Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[29] Tiobe Inc. TICS coding standards framework, 2011. www.tiobe.com.

[30] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: A data provenance perspective. In *Proc. ACM SE*, pages 68–80, 2010.

[31] R. Wettel and M. Lanza. Program comprehension through software habitability. In *Proc. ICPC*, pages 231–240, 2007.