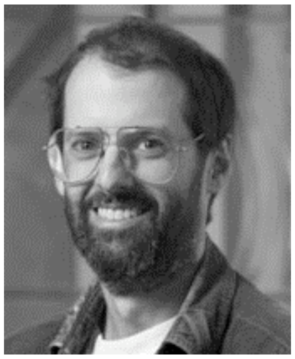
Source Code Analysis: A Road Map

David Binkley



Dr. David Binkley is a Professor of Computer Science at Loyola College in Maryland. He received is doctorate from the University of Wisconsin in 1991 at which time he joined the faculty at Loyola. From 1993 to 2000 Dr. Binkley worked as a faculty researcher at the National Institute of Standards and Technology (NIST). During this time he was part of the team that worked on the C program slicer Unravel and also considered the problems of creating high integrity (safety critical) software systems and software engineering technology transfer.

Dr. Binkley present NSF funded research focuses on improving semantics-base software engineering tools. Recently he has also been involved in a seven school collaborative project aimed at increasing the representation of undergraduate women and minorities in computer science. In 2006 he co-chaired the International Conference on Software maintenance. and in 2002 he was the general chair of the International Workshop on Source Code Analysis and Manipulation and a year later served as the program co-chair.

Source Code Analysis: A Road Map

David Binkley
Loyola College
Baltimore MD
21210-2699, USA

Abstract

The automated and semi-automated analysis of source code has remained a topic of intense research for more than thirty years. During this period, algorithms and techniques for source-code analysis have changed, sometimes dramatically. The abilities of the tools that implement them have also expanded to meet new and diverse challenges. This paper surveys current work on source-code analysis. It also provides a road map for future work over the next five-year period and speculates on the development of source-code analysis applications, techniques, and challenges over the next 10, 20, and 50 years.

1 Introduction

The tremendous increase in the amount of software in use each year produces a growing demand for programmers and programmer productivity. Hiring additional programmers is costly and ineffective if the system under consideration cannot be broken down into pieces. Given the complexity of modern software, a more viable solution is tool support. Of growing interest are tools based on *source-code analysis*. Such tools provide information to programmers that can be used to coordinate their efforts and improve their overall productivity.

From one perspective, source-code analysis is a means to an end. Under this view, the end is of paramount importance. It first must be defined. Only then can techniques for analyzing the source code in order to achieve this end be considered. Well over half of the FoSE papers within this volume represent ends for which source-code analysis plays an important role.

The other end of the spectrum is captured by the saying "if you build it, they will come." Those who hold this perspective believe that a tool capable of extracting generally useful information from the source code will find application. One advantage of this approach occurs when the information developed finds unforeseen application. For example, when working on making really small quantum dots, Michael Bowers, a Vanderbilt graduate student, was recently surprised when rather than a dull blue light, a bright white glow covered his work table [22]. This unexpected discovery is expected to replace the incandescent light bulb.

Regardless of perspective, a clear definition of source-code analysis is needed to guide the discussion herein.

> *Source code analysis* is the process of extracting information about a program from its source code or artifacts (*e.g.*, from Java byte code or execution traces) generated from the source code using automatic tools. *Source code* is any static, textual, human readable, fully executable description of a computer program that can be compiled automatically into an executable form. To support dynamic analysis the description can include documents needed to execute or compile the program, such as program inputs.

1.1 Why *Source-Code* Analysis

As the first compilers were being introduced, *real*^{tm} programmers would compile their code and then "tweak" the output assembler code. Sometimes this "tweaking" was to improve performance. Other times it was simply to reflect a personal preference of the programmer. Once tweaked, future updates that might be better made on the high-level source require one of three choices: (1) changing the high-level source, recompiling, and re-tweaking, (2) performing the change on the lower-level assembly code (a rather cumbersome task), or (3) abandoning the tweaking. Human nature being what it is, in time the final option won out (with the help of improved compiler technology and faster hardware).

Paralleling early software development, modern software projects often begin with the construction of models (*e.g.*, using the UML). These "higher level" models support, among other things, declarative testable properties that can be automatically proven. Eventually, these models are "compiled" to a "lower level" representation: source code. Unfortunately, this source code is incomplete and thus requires "tweaking." At present, both representations are maintained in parallel for awhile, but time pressures lead inevitably to an exclusive focus on the source code. Human nature being what it is, two separate formalizations will never be maintained for long.

Until such models (or some other replacement) are fully executable, the source, despite its flaws such as opaqueness, size, and intertwined ideas, is definitively "the truth," "the system," or at least the only place where the question "what does this program mean?" can be answered with certainty.

1.2 Scope

Included in the scope of the paper are both static and dynamic source-code analysis. In both cases, the extracted information must be consistent with language semantics and should help a programmer gain insight as to the source code's meaning. Such information is often described as a model; thus, source-code analysis can be viewed as model extraction followed by state-space exploration.

The extracted information should be *higher order* in that it moves away from lexical concerns and conveys abstract semantic information. Often times this information (*e.g.*, points-to information) serves as the input to other downstream analyses, tools, or techniques.

Two restrictions are used to bound the scope of the techniques considered herein. The first restriction is the assumption that the domain of application is software engineering, thus removing from consideration topics such as compiler theory, formal language theory, parallelism and parallelizing code, and migration. The second restriction is that the output not be source code. This draws a line between source-code analysis and source-code manipulation (*e.g.*, transformation). This restriction removes from consideration program translation, restructuring, and refactoring systems and other systems for source-code modification. It also rules out some things traditionally thought of as source-code analysis: for example, program slicing [15, 12, 115], which extracts a computation that is potentially scattered throughout a program from intervening irrelevant statements. Slicing has been proposed for use in many areas of software analysis [77, 69, 32, 91, 56, 11, 20, 42, 51]. However, its output is a program.

1.3 Outline

The remainder of this paper is organized as follows: First, the stages of a typical source code analysis is considered. This is followed by a brief look at current source-code analysis techniques. The core of the paper, in Sections 4 and 5, considers the future of source-code analysis. The paper concludes with a brief summary.

The remaining sections include many observations based on conversations with researchers and practitioners active in the field. Often their comments were provided "off the record" as they could not be asked to obtain corporate approval or were providing intuitions rather than a well researched positions. When possible, these comments are further documented using publicly available published works.

2 Anatomy of a Source Code Analysis

This section describes the three components of a source-code analysis: the parser, the internal representation, and the analysis of this representation. The first part parses the source code into one or more internal representations. In essence this step converts the concrete syntax into an abstract syntax better suited to a particular analysis. Most parsers are compiler-based and process the entire language [27, 35]. When only a subset of the concrete syntax must be processed, lighter-weight techniques can be applied. For example, an *island grammar* [87] allows portions (islands) of the concrete syntax to be parsed while ignoring the remainder. Parsing is the necessary evil of most source-code analysis. While not theoretically difficult, the complexities of modern programming languages, in particular those that are not LR(1) [40] and those incorporating some kind of preprocessor, significantly impede source-code analysis.

The second part of a source-code analysis is the internal representation. Many internal representations find their roots in compiler technology. They abstract a particular aspect of the program into a form more suitable for automated analysis. In essence, an abstraction is a sound, property-preserving transformation to a typically smaller domain (for example, replacing values with their types). Some internal representations are produced directly by the parser (*e.g.*, the control-flow graph), while others require the result of prior analyses (*e.g.*, dependence graphs require prior points-to analysis).

There are almost as many internal representations as there are source-code analyses. Some classic examples include the *control-flow graph* (CFG), the *call graph*, and the *abstract syntax tree* [40]. Other more popular internal representations include *static single-assignment (SSA) form*, which modifies the control-flow graph such that every variable is assigned exactly once thus making def-use chains explicit [31, 44]. SSA form simplifies and improves the precision of a variety of data-flow analyses. The *value dependence graph* (VDG) improves (at least for some analyses) on the results obtained using SSA form. The VDG represents control flow as data flow and thus simplifies analysis [114].

Two- and three-valued *logical structures* that map problems into a collection of predicates that produce one of two or one of three values are also used [104]. The predicates abstract features of the program (*e.g.*, $p(v)$ might represent the predicate "*does p point to v*"). The three-valued approach allows an analysis to return true positive results, true negative results, and a collection with unknown status. The two valued approach must include the unknown with one of the other two.

Finite-state automata have been used to represent analysis of event-driven systems and the transitions in distributed programs where they provide an excellent formalism for the abstraction of program models [106].

Finally, in support of interoperability, some internal representations are "internal" to the entire analysis, but external to the individual tools used in that analysis. Examples include XML and srcML [26].

Dynamic analysis often makes use of a *trace file*. Simply recording a complete trace takes considerable space. Ball and Larus describe an algorithm for instrumenting a program introducing minimal overhead [5]. Their approach concisely captures an execution history and thus a program's dynamic control flow while greatly compressing the trace. Other systems capture the state of the computation by sampling features that can be later used with statistical analysis techniques.

The most common internal representation is the *graph* (especially if its degenerate forms such as the tree are included). Many other "higher-level" representations are represented internally as graphs. For example, constraint systems and logical structures can be efficiently represented as graphs [38, 104]. One widely used graph is the *dependence graph*. The statement-level version (in which vertices represent the statements and predicates of the program) was introduced in work with parallelizing and highly optimizing compilers [39]. These graphs have since been used in many other analyses [60, 59, 6]. A related graph, the *Module Dependency Graph*, used by the Bunch tool, represents programs at a coarser level of granularity. Its vertices represent the modules of the system and edges the dependencies between them [79].

Other example graphs include *dynamic call graphs* [95, 92] and *XTA graphs* built in support of dynamic reachability-based interprocedural analysis [95]. These techniques are required to analyze languages such as Java that include dynamic class loading. Finally, the *Trace Flow Graph* is used to represent concurrent programs [25]. It consists of a collection of CFGs with additional vertices and edges to represent inter-task control flow.

The third and final part of a source-code analysis is the actual analysis. Analyses can be classified along six dimensions: static versus dynamic, sound versus unsound, safe versus unsafe, flow sensitive versus flow insensitive, context sensitive versus context insensitive, and complexity. A slightly different classification that focuses on object-oriented analysis is discussed by Ryder [102].

Static analysis does not account for program input; thus the result must be applicable to all executions of the program. This inevitability forces approximations to be made. In contrast, *dynamic* analysis takes program input into account (typically a single input). This allows greater precision; however, the results are only guaranteed to be correct for the particular input. Some techniques sit in between. They take into consideration a collection of initial states that, for example, satisfy a predicate.

The notion of algorithm soundness originated in mathematical logic where a deductive system is sound with respect to a semantics if it only proves valid arguments. A *sound* analysis makes correctness guarantees; thus, the output of a sound static analysis is guaranteed to be valid for all executions of the program. The output of a sound dynamic analysis is guaranteed to be valid for the execution of the program on which it was collected. *Unsound* analyses make no such guarantees, but can often quickly produce correct or "close enough to be useful" results. While a sound analysis provides guarantees, it often does so at the cost of precision. For example, sound analyses typically produce a larger number of false positives. Finally, unsound analyses can exploit "information that is unavailable to sound analyses" [62]. Examples of this kind of information include information present in comments and identifiers. Ratiu and Deißenböck describe how to exploit non-structural information such as identifiers in maintaining and extracting the mapping between the source code and real world concepts, which can help detect semantic defects such as (logical) redundancies [97].

Static analysis is traditionally described as *safe*, meaning that the answer is precise on "one side." For example, a reaching-definitions computation can determine that certain assignments definitely do not reach a given use, but the remaining assignments may or may not reach the use. This output thus represents a safe approximation to the set of all reaching assignments. Sagiv et al. present a static-analysis technique based on a three-valued logic [104], which does "one better" by capturing indecision as a third value. Thus again using reaching-definition as an example, a definition could be labeled "reaches," "does not reach," or "might reach."

A *flow-sensitive* analysis considers a program's flow of control; thus, given the sequence `p = &a; q = p; p = &b`, a flow-sensitive points-to analysis can determine that `q` cannot point to `b`. In contrast a *flow-insensitive* analysis treats the statements of a program as an unordered collection and must produce conservative results that are safe for any order. In the above example, a flow-insensitive points-to analysis must include that `q` might point to `a` or `b`. This reduction in precision comes with a reduction in computational complexity.

Context affects interprocedural analysis. A *context-sensitive* analysis respects the stack model of procedure call and return. Thus, when procedure P is called from call-site c_1, results from the analysis of P are propagated (only) back to c_1. In contrast, a *context-insensitive* analysis would propagate the result from P back to all call-sites on P. Context-insensitive analysis need only maintain a single approximation for each procedure, but the summary must be safe for all calls, which lowers the precision of the information.

Finally, most analysis problems have a spectrum of solutions that represent precision-effort trade-offs. For example, imprecise points-to sets can be computed in near linear time, while the computation of flow- and context- sensitive points-to sets is NP-hard [72].

The third part of a source-code analysis includes analyses used to build the representations discussed at the beginning of this section, those used in the applications discussed in the next section, and those described in Section 4. As an example, reachability analysis is a common analysis [98, 13]. Questions such as "Can procedure P (transitively) call procedure Q?" can be framed as graph-reachability questions on the program's call-graph. Similarly, the question "Does the value assigned to `x` at statement `s` reach a use of `x` at statement `t`?" begins with the reachability question "is there a path from `s` to `t` in the program's control-flow graph?"

The remainder of this section considers points-to analysis as a representative example of a source-code analysis [72, 23, 116, 100]. Points-to analysis was chosen as it is a challenging problem that is an essential precursor to many other source-code analyses. Traditionally, points-to analysis techniques are partitioned into four groups depending on their flow and context sensitivity.

Three algorithms are considered as representative examples. The first two have the same precision and are flow- and context-insensitive while the third is flow and context-sensitive. The first two read in normalized assignment statements output by the parser (the first part of the analysis). These statements, which represent the pointer manipulations in the program, have one of four forms.

$$\begin{array}{ll} p = q & p = *q \\ p = \&q & *p = q \end{array}$$

The first algorithm is Andersen's graph closure-based algorithm [1]. The graph built by Andersen's algorithm consists of vertices that represent memory locations and edges that represent relations between vertices. Vertices are created for variables, addresses of variables, and dereferences of variables. The graph contains four kinds of edges (named A, G, R, and W) that are defined as follows:

1. $p \xrightarrow{A} q =_{df}$ q is in the points-to set of p.
2. $p \xrightarrow{G} q =_{df}$ points-to(q) $\subseteq$ points-to(p).
3. $p \xrightarrow{R} q =_{df}$ p represents the dereference of q
4. $p \xrightarrow{W} q =_{df}$ p represents the dereference of variable x and $*x = q$.

The initial graph fragments for each normalized assignment are as follows.

$$\begin{array}{ll} p = q & p \xrightarrow{G} q \\ p = *q & p \xrightarrow{G} *q \xrightarrow{R} q \\ p = \&q & p \xrightarrow{G} \&q \xrightarrow{A} q \\ *p = q & *p \xrightarrow{R} p \text{ and } *p \xrightarrow{W} q \end{array}$$

After construction of the initial graph, the following rules are applied until a fixed point is reached.

$$\begin{array}{ll} \text{Rule 1:} & a \xrightarrow{G} b \xrightarrow{A} c \Rightarrow a \xrightarrow{A} c \\ \text{Rule 2:} & a \xrightarrow{R} b \xrightarrow{A} c \Rightarrow a \xrightarrow{G} c \\ \text{Rule 3:} & a \xrightarrow{G} b \text{ and } a \xrightarrow{W} c \Rightarrow b \xrightarrow{G} c \end{array}$$

For example, Rule 1 states that if points-to(b) $\subseteq$ points-to(a) and c is a member of points-to(b) then c is also a member of points-to(a) (*i.e.*, add the edge $a \xrightarrow{A} c$ to the graph). The final points-to set for variable v is extracted from the closed graph as the set of all variables labeling vertices reachable via an $\xrightarrow{A}$ edge from the vertex for v.

The second algorithm is based on the work of Fahndrich et al. [38]. Their analysis is actually a general constraint solver. The following description is specialized to the problem of pointer analysis. Unlike the Andersen algorithm, the Fahndrich algorithm, does not compute a complete closure of the graph. Rather, the points-to set for a variable v is computed "on the fly" as needed.

The approach manipulates a graph that involves *"ref"* structures, which are used to represent dereferencing and taking the address of variables. The graph can be thought of as having three parts (often drawn as columns). Column 1 entries, denoted by *ref*(V, V), represent the addresses of variables. Column 2 entries, denoted simply as V, represent variables. Column 3 entries represent two forms of dereferenced variables. Dereferences on the right-hand side of an assignment are denoted by *ref*$(V, \emptyset)$, while left-hand side dereferences are denoted by *ref*$(1, V)$, where "1" represents the universal set.

The Fahndrich algorithm finds paths from Column 1 entries to Column 3 entries. Take, for example, a path from Column 1 entry *ref*(V, V) to Column 3 entry *ref*(A, B). In terms of the constraint solver, this path is equivalent to the constraint *ref*$(V, V) \subseteq$ *ref*(A, B). Because both terms are *ref* terms, the corresponding arguments yield additional constraints. These constraints differ in that the first argument of a *ref* is *contravariant*, while the second is *covariant* [38]. The contravariant constraint generates the new constraint $V \subseteq A$, while the covariant constraint generates the new constraint $B \subseteq V$. These new constraints are represented as new edges in Column 2 of the graph.

In the case of pointer analysis, the algorithm repeats the following process until there are no changes. For each entry *ref*$(1, V)$ in Column 3, let P represent its points-to set (*i.e.*, entries from Column 1 that have a path to *ref*$(1, V)$). For each such entry *ref*(U, U) in P, add an edge (in Column 2) from V to U. This edge arises from the constraints $U \subseteq 1$ and $V \subseteq U$. Next, for each entry *ref*$(V, \emptyset)$ in Column 3, let Q represent its points-to set. For each entry *ref*(U, U) in Q, add an edge (in Column 2) from U to V. This edge arises from the constraints $U \subseteq V$ and $\emptyset \subseteq U$. When no changes occur, the points-to set for each variable V contains all the variables A for which there is a path from *ref*(A, A) to V. It is obtained by walking edges backwards from V and gathering up vertices from Column 1.

Fahndrich et al. describe four optimizations that dramatically improve the constraint solver's efficiency. Two are essential for pointer analysis. The first collapses cycles found in Column 2. Cycles occur when a set of variables must all have the same points-to set: for example, when $A \subseteq B \subseteq C \subseteq A$. Collapsing such cycles to a single vertex avoids significant redundant work.

The second optimization caches points-to sets associated with Column 2 vertices. The cache is invalidated on the start of each iteration of the main loop. Note that within an iteration the cache may be out of date (always a subset of the actual points-to set). This only occurs as the result of a change; thus, an updated value will be computed and used on subsequent iterations. In general, both cycle collapsing (strongly-connected component collapsing) and caching (memorization) improve the efficiency of graph-based analysis algorithms [13].

The third example point-to algorithm, described recently by Hackett and Aiken, performs flow- and context- sensitive analysis [47]. The algorithm is summary-based following in the line of Liang and Harrold, and others [75, 36, 73]. To obtain context sensitivity, a meta-level description of the aliasing impact of each function is maintained. Consider, for example, two calls to the identity function such as a = id(p) and b = id(q). Rather than summarizing the output alias state as a concrete collection of aliased locations, a single graph is used to describe how an input aliases configuration is mapped to an output configuration.

This algorithm exploits patterns in the alias relationships (discovered through empirical study) to achieve an order of magnitude performance improvement over past algorithms. Four levels of summary information are maintained: global, type, function, and local. The savings come from storing information at the highest level possible. For example, consider a binary tree with parent pointers. At the local level, each tree node must be associated with the collection of locations that are potential parents. Using a type level pattern, these alias relationships can be summarized once using pattern aliases(tree.parent, tree).

Flow sensitivity is supported by labeling edges in the graph with boolean predicates that describe intraprocedural control-flow paths. For example, $\mathsf{b} \xrightarrow{B} \mathsf{a}$ says that b points to a provided B is true. The labels require boolean satisfiability; however, as the algorithm only requires this on an intraprocedural level it is within the capability of existing theorem provers as function size is relatively constant regardless of program size.

Looking ahead to Section 5, this work is an excellent example of an emerging pattern in computer science where, "better" takes on a statistical sense. For instance branch prediction does not guarantee performance improvement it is only statistically likely based on empirical study of typical programs.

In this case, although flow- and context-sensitive points-to analysis is in general NP-hard, it becomes possible to efficiently solve a subset of the problem, namely that found in existing C code. Hackett and Aiken found that aliasing has a great deal of structure in real programs and that just nine programming idioms account for nearly all aliasing found in their study.

3 Applications of Source Code Analysis

From its beginning in the compiler community, the use of source-code analysis has spread into a variety of software engineering tasks. This section lists many of these applications. Those appearing in italics are covered in more detail after the list as representative examples; those marked with a † are considered elsewhere in this volume.

- architecture recovery [8, 105, 21]
- assertion discovery [37]
- automotive software engineering† [94]
- *clone detection* [80, 71, 21]
- comprehension [19, 28, 101, 21]
- *debugging* [108, 90, 45, 24]
- empirical software engineering research† [110]
- fault location [83, 64]
- middleware† [61]
- model checking in formal analysis† [34]
- model-driven development† [41]
- optimization techniques in software engineering† [48]
- performance analysis† [117]
- program evolution [9, 118]
- quality assessment [107, 66]
- *reverse engineering*† [21]
- safety critical† [54]
- software maintenance [52, 30]
- software reliability engineering† [78]
- software versioning [112, 43]
- specification semantics [113]

- symbolic execution [2, 67]
- testing† [10, 53]
- tools and environments† [119]
- validation (conformity checking)
- verification, sound formal [18]
- verification, unsound syntactic [17, 29, 65]
- *visualizations of analysis results* [14, 93, 99, 4, 16]
- web application development† [63]

Debugging. The first representative example is debugging, which is one of the hardest, yet least systematic activities in software engineering. The popularity of debugging and debuggers as a research topic has waxed and waned over the years. Perhaps this is due to the ever increasing complexity of the problem imposed by steadily more complex compiler back ends and new language features (*e.g.*, reflection). In many ways debugging technology has lagged behind the advances in language design and other software development tools such as IDEs. Some recent debugging innovations that counter this trend include algorithmic debugging, delta debugging, and statistical debugging.

Algorithmic debugging uses programmer responses to a series of questions generated automatically by the debugger. There are two clear goals for future algorithmic debuggers: first, reducing the number of questions asked in order to find the bug, and second, reducing the complexity of these questions [109]. For example, the first of these is possible by automatically generating heuristics for asking and answering higher-level questions.

Delta debugging systematically narrows the difference between two executions: one that passes a test and one that fails. This is done by combining states from these two executions to automatically isolate failure causes. At present the combination is syntactically defined in terms of the input, but a more sophisticated combination might use dependence information to narrow down the set of potential variables and statements to be considered.

Finally, the Sober tool uses statistical methods, akin to hypothesis testing, to automatically localize software faults [76]. Using predicate execution patterns from correct and incorrect executions, Sober identifies predicates that have statistically different behaviors in the two executions. These are output as fault-relevant predicates.

Clone detection The second representative example, clone detection, is typically defined as identifying similar sequences of code (according to some similarity measure) [21]. Many different algorithms for detecting clones have been proposed. They use a variety of source-code analyses. For example, some are based directly on the program text. These include techniques that compare whole files, strings, substrings, or identifiers (*e.g.*, using latent semantic indexing). More structured techniques compare tokens (*e.g.*, using suffix trees or parameterized strings). Some recent approaches have used more powerful source-code analyses including dynamic programming, data mining, program dependence graphs, and execution traces.

Reverse Engineering. Reverse engineering is an attempt to analyze (typically) source code to determine the know-how which has been used to create it [21]. Pattern-matching approaches in reverse engineering aim to incorporate domain knowledge and system documentation in the software architecture extraction process. Most existing approaches focus on structural relationships (such as the generalization and association relationships) to find design patterns. However, behavioral recovery, a more challenging task, should be possible using data mining approaches such as sequential pattern discovery. This is useful as some patterns are structurally identical but differ in behavior. Dynamic analysis can be useful in distinguishing such patterns.

3.1 Two Applications on the Fringe

The definition of source-code analysis laid out in Section 1.2 focuses the paper on techniques that do not output source code. This was done to remove from consideration techniques such as transformation and refactoring. However, it also draws the paper away from several source-code analysis techniques that produce "throw way code." This section briefly considers two such approaches.

The first throw-away code example generates code used in testability transformation, a form of program transformation in which the goal is not to preserve the standard semantics of a program, but to preserve test sets that are adequate with respect to some chosen test adequacy criterion [7]. The transformed program is used in the test-case generation process and then discarded. Thus, it is merely a "means to an end," rather than an "end" in itself. One advantage of this approach is that the transformation process need not preserve the traditional meaning of a program. For example, in order to cover a chosen branch, it is only required that the transformation preserve the set of test–adequate inputs for the branch.

The second throw-away code example, "programs as answers," begins with the observation that "the most important requirement imposed by applications in software engineering results from the fact that programmers are faced with the result of the analysis. They should be able to understand it and relate it to the source program [85]." Thus, when the consumer is a programmer, one can easily advocate using source code as the "language" of the result. For example, source has been used to communicate partial answers to undecidable problems [50]. The underlying philosophy of this approach is that undecidable hypotheses can be "answered" by a partly automated system which returns neither "true" nor "false", but rather a program that computes the answer. To answer a question the program source is first augmented to make the question explicit in the code. An amorphous slice [49] taken with respect to the output of the augmentation produces a program that computes the property of interest. The resulting test program is significantly simpler than the original program, thereby simplifying the process of investigating the original hypothesis.

A symmetric idea is to use source as the input to a tool (not the input being analyzed but the input controlling the analysis). For example, Martin et al. present a query tool in which a query is in essence a code excerpt corresponding to the shortest amount of code that would violate a design rule [81].

4 Current Research Challenges

This section first considers three general principles that guide future research work on source-code analysis and then describes a collection of relevant research areas.

Principal 1: Power Hungry is Good. Techniques that require significant processing power (in terms of CPU cycles and memory) will benefit from increased processing power. While there is doubt regarding the continuance of Moore's law, for the next three to eight years the trend is expected to continue [88]. Of particular value to many analyses is an increase in available memory. Classic source-code analyses, such as points-to analysis, typically run out of space before running out of time. Unfortunately, given present code size, Moore's Law would need to hold for some time before existing precise solutions to many data flow problems become feasible for larger programs [89]. In the interim, algorithmic improvement and more intelligent management of the memory hierarchy are needed. The recent alias pattern discovery work of Hackett and Aiken is an excellent example of such management [47].

One class of power-hungry techniques that will continue to benefit from increases in processing power are genetic algorithms [82, 48]. An example of this approach is the Bunch tool [86], which uses search to perform software clustering. Its output is an architectural-level view of a system's structure obtained directly from the source code. The current tool considers a program at the class level. A factor of ten increase in computing power would allow finer grained analysis (*e.g.*, the method level); thus, improving the precision of the modularization presented to the engineer.

Principal 2: Less is More. Does this movement that started in graphic design speak to source-code analysis? Any programmer who has set out to write an analysis tool confronts significant obstacles. First, simple parsing is not as simple as it should be. For example, C++ is not $LR(n)$ (for any n!). Furthermore, the presence of casting, pointer arithmetic, dynamic class loading, reflection, and the like, complicate semantic analysis. This leads to the question "how good a tool would I need to create for programmers to use a 'simple' language?" Language design seems headed in the opposite direction (Fortran "9x" is an excellent example). Against an increasing need for higher precision source-code analysis, modern languages increasingly require tools to handle only partially known behavior (in the case of Java this is caused by features such as generics, user-defined types, plug-in components, reflection, and dynamic class loading). These features increase flexibility at runtime, but compromise static analysis. In considering each of the following research areas, consider also the role that analysis quality should play in the language design debate.

Principal 3: Of C3P0 and Han Solo. The tasks that machines tend to be good at are tedious bookkeeping tasks. In contrast, programmers are better able to make "ah ha" type discoveries. This suggests the use of semi-automatic analysis where peak performance is obtained through a symbiotic relationship between the two. However, programmers can be quickly overrun (for example, by too many false positives). Effective techniques for the problems described in this section may involve creative use of the programmer in providing key human insight.

The remainder of this section describes a collection of research areas that have seen recent interest or are expected to over the next three to eight years. It considers these analyses in a rough progression from those that are established, but continue to need work, through emerging techniques that should receive additional attention, and finally toward "new" ideas that deserve exploitation.

Pointer analysis. Pointer analysis is the archetypical example of a source-code analysis. The results are essential to many other down-stream analyses, yet after years of work the problem is still "unsolved" [57]. Research has produced improvements; unfortunately, the problem is complex [58] and thus warrants further research.

Many down-stream analyses need the precision of a flow- and context-sensitive points-to analysis. One approach to providing this precision is to factor in the requirements of the down-stream analysis [46]. This allows the points-to analysis to focus on those facets of the problem of interest to the down-stream analysis at not more than the necessary level of precision.

Concurrent Program Analysis. Current trends in hardware design (*e.g.*, dual core processors) make it clear that future processing power will come from multiple processing units. This increases the need for analysis tools that can help programmers author, maintain, comprehend, and otherwise manage concurrent programs. It also suggests that successful future source-code analysis algorithms will be those designed to run concurrently.

An example technique that can exploit significant parallel processing power is symbolic execution. For example, consider its applications to automatic test-input generation. Recent advances in decision procedures coupled with increases in processor speed make it possible to consider interprocedural symbolic execution. The challenge in doing so is the exponential number of program paths to be considered. Abstraction techniques can reduce the number of paths. This reduction can be based on a program specification or on problem-specific abstraction. For example, to generate test inputs for branch coverage, a rather simple technique removes all program paths that do not include a branch that needs to be covered. Other examples include the analysis of distributed concurrent programs including internet (web) centric applications.

Dynamic Analysis. Another technique that can exploit increased (parallel) processing power is dynamic analysis. Multiple processors also raise the possibility of real-time dynamic analysis. Thus, while the program is being written or debugged, speculative execution of the program can be used to gather information that a programmer could use. One example is the dynamic analysis and test of COTS component-based systems where techniques automatically derive behavioral models by monitoring component executions and then dynamically check these models to detect possible misbehaviors and incompatibilities when the components are replaced with new components or used as part of new systems.

Recent research has included investigations with unsound techniques. (The output of a sound analysis is guaranteed to be valid. Unsound analysis makes no such guarantees.) For example, the Daikon tool makes use of multiple executions of a program to discover potential program invariants (properties that always hold at a given point in the program) [37]. The technique is unsound in that the assertions are not guaranteed to be valid on all program executions. However, in practice, Daikon has proven quite useful. This kind of analysis is one that can easily exploit multiple processors in the multiple executions of the program. Future research on source-code analysis should consider other (unsafe) (dynamic) analysis techniques that combine results from multiple executions.

Analyzing Executables. The static analysis of executables can be used to recover intermediate representations of quality (similar) to those produced from the source code [3]. Here analysis must be carried out to, for example, recover information about the contents of memory locations and how they are manipulated by the executable. Another example models an abstraction of the stack by using stack based instructions to statically detect obfuscated calls (primarily used by malicious code) [70]. One advantage of analyzing binaries is that library stubs are not necessary as the technique can directly analyze library code (although system calls still need to be modeled). When source is lost or missing, these techniques grow in importance.

Information retrieval. Information retrieval (IR) is a well established field that has blossomed in the last fifteen years with the growth of the Internet and the huge amounts of information available in electronic form.

Existing applications of IR techniques to source-code analysis include automatic link extraction, concept location, software and website modularization, reverse engineering, software reuse impact analysis, quality assessment, and software measurement. These applications treat code as text rather than considering its structure. For example, such techniques might estimate a language model for each "document" (*e.g.*, source file, class, error log, requirements, etc.) and then use a Bayesian classifier to score each. Much of this work has a strong focus on program identifiers. For example, IR's cosine similarity has been used to rate the relative quality of modules from within a program producing a semi-automatic quality assessment [74]. Unlike other approaches that consider non-source code documents (*e.g.*, the requirements), this approach focuses exclusively on the code. It divides each source code module (currently a function) into two documents: one includes the comments and the other the executable source code. The cosine similarity between the two is measured and used as a proxy for quality. Empirical evidence supports this use. The technique fills the gap where automated techniques have been found lacking and where direct human assessment is prohibitively expensive.

To date, the application of IR has concentrated on processing the text from source and non-source software artifacts (which can be just as important as source) using only a few well-developed IR techniques. Given the growing importance of non-source documents, source-code analysis should in time, develop new IR-based algorithms specifically designed for dealing with source code.

Data Mining. The mining of software-related data repositories has only started. It represents a shift to "discovery-driven" analyses, which sift through large amounts of data and automatically (or semi-automatically) discover important information [84]. These techniques require significant computing resources and the application of techniques such as pattern recognition, neural networks, association measures, and decision trees, which have advanced dramatically in recent years.

Data mining techniques include classification trees, Bayesian belief networks, clustering, and statistical tests. For example, classification trees have been one of the tools of choice for building classification models in software engineering. They are built by selecting attributes (one at a time) from the data. Branches are introduced for each value the selected attribute can have. The algorithm for building classification trees seeks to find *attributes* within the data that provide maximum segregation of data records at each level of the tree. Trees are evaluated based on training data and refined (*e.g.*, to improve the order of the attributes). In this way a classification tree can be used to discover classification rules for a chosen attribute of a data set by systematically subdividing the information contained in the data set.

Most existing research has been conducted by software engineering researchers, who often reuse simple data mining techniques such as association mining and clustering. A wider selection of data mining techniques should see more general application that removes the requirement that existing systems fit the features provided by existing mining tools. For example, API usage patterns often involve more than two API method calls or involve orders among API method calls, leaving mining for frequent item sets insufficient. Finally, the mining of API usage patterns in development environments as well as many other tasks pose requirements that cannot be satisfied by reusing existing simple miners in a black-box way. Thus, there is demand for the adaptation or development of more advanced data mining methods.

Multi-Language Analysis. Multi-language analysis grows more important as (primarily web-based) systems are built of many parts composed of many languages [111]. Of particular challenge are large systems that contain different languages and domains.

The Confluence of Static and Dynamic Analysis. The simplest combination (akin to parallel play) is a tool that uses both static and dynamic analysis to achieve a goal. For example, Martin et al. describe an error detection tool that checks if a program conforms to certain design rules [81]. This system "automatically generates from a query a pair of complementary checkers: a static checker that finds all potential matches in an application and a dynamic checker that traps all matches precisely as they occur." The static analyzer finds all potential matches conservatively using a context-sensitive, flow-insensitive, inclusion-based pointer alias analysis, while the dynamic analyzer instruments the source program to catch all violations precisely as the program runs.

Slightly more sophisticated combinations often use static analysis to limit the need for instrumentation in the dynamic analysis. Path testing tools use this approach as does Martin et al.'s error detection tool, where "static results are also useful in reducing the number of instrumentation points for dynamic analysis." They report that the combination proves able to address a wide range of debugging and program comprehension queries.

In the "other direction," Gupta et al. present an algorithm that integrates dynamic information from a program's execution into a static analysis [45]. The resulting technique is more precise than the static analysis and less costly than the dynamic analysis.

Heuzeroth et al. consider the problem of reverse engineering design patterns using a more integrated (though sequential) combination of static and dynamic analysis [55]. In this case, static analysis is used first to extract structures regarding potential patterns and then dynamic analysis verifies that pattern candidates have the correct behavior. Here the static analysis does more than improve the efficiency of the dynamic approach. The two truly compliment each other.

Closer still to true integration is a combination that, in essence, iterates the two to search for test data input. This technique applies a progression of ever more complex static analysis with (genetic-based) search. This synergistic arrangement allows low-cost static analysis to remove "obvious" uninteresting paths. It then applies relatively naive, but inexpensive dynamic search. If more test data is needed, more sophisticated static and then dynamic techniques are applied. All these techniques, however, fall short of a truly integrated combination of static and dynamic techniques. Future combinations should better integrate the two.

Non-functional properties. The trend towards portable computing brings back issues such as memory footprint and compiled code speed. While these issues have faded in importance in the context of modern desktops, the rising use of portable and embedded computing has caused them to cycle around again. In addition, portable computing also raises issues such as power consumption. This raises the possibility of analysis aimed at limiting the number of functional units that must be active.

Self Healing Systems. Corrupt data structures can cause unacceptable program execution. Data structure repair (which eliminates inconsistencies) can enable a program to continue to execute acceptably in the face of otherwise fatal data structure corruption errors [33]. A recent approach to achieving dependable systems is to incorporate "self-healing" real-time data structure repair. A taxonomy of such techniques is provided by Koopman [68]. While no consensus-based definition of the term "self-healing" exists, intuitively, these systems automatically repair internal faults. Relevant aspects of self-healing systems include fault models, system responses, system completeness, and design context.

An example of a self-healing system was studied recently by Demsky et al. [33]. This system identifies and corrects corrupt data structures (a common cause of abnormal program termination) while a program is running. This enables a program to continue execution in the face of otherwise fatal data structure corruption. The Demsky et al. approach generates candidate data-structure consistency rules, which are reviewed by an engineer.

The need for high-integrity safety-critical software is one force that drives the need for continued research on self-healing code: for example, there is a need for the application of static analysis techniques to failure identification and to the final assessment of the healing mechanisms. Source code analysis should also see a growing role in the design of self-healing applications, such as the automation of certain self-healing activities.

Real-Time Analysis. The final research problem considered has two distinct facets: compile time and run time. Self-healing code and instrumented code are run-time examples. Here analysis is being done real time while the program is executing. The archetypical example of this idea is just-in-time compilation.

With increased processing power, real-time analysis will also work its way into compile time. IDE's that perform syntax coloring and simple syntactic error checking perform a limited form of this kind of real-time analysis.

Looking forward, more such processing can be done in real time. For instance, code coverage and memory-leak analysis might be performed, at least partially, at compile time instead of at run time. This has the advantage of providing information about a piece of code that is the current focus of the programmer. Such analysis is already being integrated into IDEs, a trend that should continue.

5 Future Challenges

This section speculates on the future of source-code analysis. It first considers the state of the art ten, almost twenty, and then fifty years hence. While increased processing power (primarily CPU cycles and memory) make certain things possible, the computational complexity of many interesting source-code analysis problems (high-degree polynomial or exponential) greatly reduce these benefits. Thus research on improving source-code analysis algorithms continues to be a need.

5.1 Ten Years Hence

In ten years the global code base will approach 500 billion lines of code (40-60% of which will be Cobol). When analyzing this code, space will continue to be more of a problem than time. Furthermore, analysis aimed at reducing memory footprint, power consumption, and execution time especially for nano and small communicating devices will continue to grow in importance. This section considers the impact of these technology advances and of the increased reliance on software, on source-code analysis ten years into the future.

While at present there exists a rather polar debate on the value of formal methods, as engineering discipline grows within software engineering, this discussion will shift from a "yes" or "no" discussion to a measure of degrees. For example, most motorists think of the bridge they are driving over as safe, but the engineer who designed it thinks of it as "safe enough." It could have been safer, but certain engineering tradeoffs were made, most involving time-to-completion, materials available, and cost.

The proliferation of software, in particular in embedded and safety-critical systems, will drive the need for greater engineering discipline, although other domains, such as information security, will also play a role. This will require software metrics and thus a software construction processes that allows for the measurement and prediction of software quality. It will also require greater engineering discipline to balance this increased need for software reliability against the cost effectiveness of software construction and, in particular, a continued demand to be first to market. In the end, this will drive the need for increased precision in source-code analyses.

One emerging trend that will help support this effort is the proliferation of source-code analysis throughout the coding process. This includes source-code analysis performed at edit time, compile time, link time, and run time. An example of this is moving (some) testing "up front" to be done in parallel with development. For example, Staff and Ernst suggest that "spare CPU resources to continuously run tests in the background, providing rapid feedback about test failures as source code is edited" [103]. At the same time, the demand to be first to market may result in "live testing:" testing pushed out to the consumer, with software containing intelligent agents that detect problems and produce corrections and workarounds on-the-fly. Partial tool support for this process already exists. For example, the Gamma tool performs the analysis and measurement of deployed software and allows for gathering program-execution data from the field [64].

A second example is the performing of source-code analysis at link or run time which is being driven by language features such as reflection and dynamic class loading. Another run-time example is real-time verification. Similar to existing hybrids of static and dynamic analysis, these analysis techniques will be hybrid techniques that eventually combine results from the different times. Thus, source-code analysis tools will need to use information from edit, compile, link, and run time and continue to include a combination of multiple views of a software system such as structure, behavior, and run-time snapshots.

Other techniques may emerge, such as those that combine source-code analysis with natural language analysis. At present this has started with source-code analyses that incorporate natural language through information retrieval techniques.

Future tools will also need improved support for user interaction. One way of doing this is to include the programmer "in the loop". However, rather than being asked to make a collection of similar low-level choices, tools will ask about higher-level patterns that can be used to avoid future questioning. One place where this would have a big payoff is in reducing false positives: for example, automated tools such as JTest, WebInspect, Empirix, FindBugs, and Clover, are a great help to programmers, but sifting through false positives and reproducing errors is, at present, unproductive.

The need for better tools may finally drive next generation languages to incorporate more thought on the entire programming process and not just flexibility for the initial programmer. For example, such a language might support efficient flow-sensitive analysis. Another potential growth area is in code-analysis tools that move from a semantic orientation to a service orientation (*e.g.*, predicting performance, etc.). Given enough computational power and some language design work, another way this might come to pass is through general purpose model-based programming languages. While initially the runtime code may be bloated and inefficient, increased computational power should compensate for this bloat, while the payoff will come from increased programmer productivity (reduced time, improved quality, etc). As this happens it will redefine the source-code analysis problem to be a model-analysis problem.

5.2 In the Year 2025

Looking almost twice as far into the future, in the year 2025 the global code base should top 1 trillion lines of code (less than half of which will be Cobol). Space rather than time should continue to be a larger issue in the analysis of source code. However, as the wheel of time turns, memory footprint and power consumption will again become less of an issue. This section considers the impact of further technology advances and of the increased reliance of software on future source-code analysis.

Present architectural trends do not suggest faster processors, but rather more processors. Assuming something like Moore's law continues to hold (which may be unlikely) technology will double the number of processors every 18 months. From the present two in commodity chips such as the core-duo, in 18 years there will be 8K processors in commodity-level desktop computers (to the extent that they still exist). Thus, for moderately sized programs, each function could be assigned its own processor. This supports the increased use of precise and sound algorithms. Finally, it gives an advantage to analyses that can themselves be parallelized.

Two examples of the impact this processing power will have include tools that appear to exhibit "intelligent" behavior and improved data mining. For example, increased power would enable stochastic methods of source-code analysis that use Bayesian analysis against a database of known "good" and "bad" code to identify regions of source with a higher probability of resembling something known to be problematic. Along similar lines, "source-code" analysis tools could assess the design and architecture early in a software development project, thus heading off problems related to bad design decisions and overly complex architectures.

As a second example, increased processing power should allow data mining techniques to produce good quality models based on huge amounts of data. It also allows data mining researchers to tackle problems of on-the-fly real-time mining in software development environments (*e.g.*, in IDEs such as Eclipse) based on the engineering data accumulated from the development history of a team of programmers. This can lead to tools that appear to "understand" algorithms and can automatically suggest superior solutions to specific problems.

The above examples assume that the programming process remains largely unchanged. However, programing may see a significant paradigm shift. For example, if tools such as Workflow become more prevalent, a lot less coding will be done as "programmers" simply glue services and components together into work flows. Thus, in a ratio similar to the number of bridge builders to bridge engineers, a large number of people will stitch together existing services to make "custom" applications for businesses, while a relatively small number of software engineers will develop the reusable services. This raises the need for source-code analysis tools that help program builders correctly assemble the pieces. One such testing tool might run the code millions of times with simulated inputs while it is being assembled.

A second paradigm shift would come if model-based approaches become everyone's daily approach to programming. This might happen gradually if, initially, online verification can be used to continuously check the consistency between source code and higher level models. Eventually, assuming language designers can solve questions such as how to make model-based languages expressive enough that they can be widely applied, model based languages will emerge. The source-code analysis questions that this raises include techniques for dissuading programmers from writing huge, ugly, and complex models which themselves will become legacy and need to be analyzed. Moreover, if these languages fail to completely replace the programming languages of today, there will be a mixture of models and programs that need to be analyzed together.

5.3 Fifty Years Hence

Essentially doubling the age of software engineering, in fifty years the global code base will include trillions of lines of code equivalents (the dominance of "higher-level" programming languages (*e.g.*, model-based languages) making the comparison difficult). In rather broad strokes, this section tries to envision the makeup of source-code analysis in fifty years time.

Today, source-code analysis tools can be quite costly (partially due to license agreements). For example, the list price for Coverity's Prevent [29] is 50,000 US dollars. This reflects the uncertainty of tool development and the need for continuous updating to reflect the latest language standard and the popularity of the target language. For the development of large sophisticated tools, this makes the open-source model considerably more viable.

The history of source-code analysis finds many examples of analyses that did not scale to the largest programs. As processes and technique improved, the size of the program also grew. This raises the question, "without any significant paradigm shift, how much larger can programs become?" (Assuming that loosely coupled, communicating distributed programs can be analyzed independently.) Factor in hardware improvements and flow- and context-sensitive, precise source-code analysis may become practical for the range of program sizes that programmers are reasonably capable of producing. However, if program size keeps growing, analysis will require finding a way to track and summarize pieces of systems and algorithms that can combine summary information in specific contexts to obtain a full analysis.

Given the time frame, a large paradigm shift in computing (not just programming) is possible. For example, new technologies such as quantum computing, DNA computing, or some other non-von-Neumann computing model might emerge. Today in the laboratory for certain problems, quantum computation provides exponential speed-up. If exponential processing speeds become widely available, then precise context-, flow-, and thread-sensitive safe solutions to the data-flow problems that underly source-code analysis tools can be applied to large programs. Unfortunately, no amount of hardware improvement will solve certain problems. For example, identifying context-sensitive synchronization-sensitive paths is undecidable [96]. However, such improvements will change the nature of source-code analysis tools through both the application of existing tools and by allowing new algorithms to be considered.

6 Summary

Until alternate formalisms come into common usage, source code will continue to be definitive in describing program behavior. Based on the past 40 years, programming languages can be expected to continue to incorporate new features that complicate program semantics. Given its central role in software engineering, source code and source-code analysis will remain "hot topics" and thus the focus of intense research activity into the foreseeable future. To ensure future progress including necessary, but unforeseen breakthroughs, it is important for future source-code analysis research to continue to investigate a wide diversity of ideas. This paper has set forth a number of current and future directions for such research.

7 Acknowledgments

Special thanks to Mark Harman without whom this paper would not exist in its present form. He is also credited with the idea described in Section 4 at the end of the confluence of static and dynamic analysis. Special thanks also to Rainer Koschke for his insights and information and to Alex Aiken, Jeri Hanly, Mike Hinchey, Dawn Lawrie, Phil Pfeiffer, Tom Reps, and Josep Silva for their detailed comments on drafts of the paper. Thanks also to the following for their comments and suggestions on the future or source-code analysis Saswat Anand, Paul Anderson, V. J. Benokraitis, Darren Curry, Tom Dean, Florian Deißenböck, Massimiliano Di Penta, Matt Dwyer, Michael Ernst, David Etkins, Keith Gallagher, Tibor Gyimothy, Mary Jean Harrold, Gorel Hedin, Michael Hind, Kevin Hunter, Gaspard Justin, Jens Krinke, Arun Lakhotia, Spiros Mancoridis, Tom McKearney, Phil McMinn, Tom Mens, David Notkin, Alex Orso, Mauro Pezze, Ross Raszewski, Daniel Ratiu, Barbara Ryder, Kamran Sartipi, Sibylle Schupp, Harry Sneed, Eleni Stroulia, Paolo Tonella, Jason Tyler, Tao Xie, and Xiaofang Zhang. This work is supported by National Science Foundation grant CCR0305330.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] C. M. B. Botella, A. Gotlieb. Symbolic execution of floating-point computations. *The Software Testing, Verification and Reliability Journal*, to appear.

[3] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. of the International Conference on Compiler Construction*. Springer-Verlag, New York, NY, 2004.

[4] T. Ball and S. G. Eick. Visualizing program slices. In A. L. Ambler and T. D. Kimura, editors, *Proceedings of the Symposium on Visual Languages*, Los Alamitos, CA, USA, Oct. 1994. IEEE Computer Society Press.

[5] T. Ball and J. Larus. Efficient path profiling. In *ACM/IEEE International Symposium on Microarchitecture*, 1996.

[6] F. Balmas. Using dependence graphs as a support to document programs. In 2^{nd} *IEEE International Workshop on Source Code Analysis and Manipulation*, Montreal, Canada, Oct. 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[7] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, Omni Parker House Hotel, Boston, Massachusetts, July 2004. Appears in Software Engineering Notes, Volume 29, Number 4.

[8] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice second edition*. Addison Wesley, 2003.

[9] K. Bennett and V. Rajlich. Software maintenance and evolution: A roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[10] A. Bertolino. Software testing research: Achievements, challenges, dreams. *Future of Software Engineering, L. Briand and A. Wolf (eds.), IEEE-CS Press*, 2007.

[11] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8), Aug. 1997.

[12] D. Binkley and K. B. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances in Computing, Volume 43*. Academic Press, 1996.

[13] D. Binkley and M. Harman. Results from a large–scale study of performance optimization techniques for source code analyses based on graph reachability algorithms. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, Amsterdam, Netherlands, Sept. 2003. IEEE Computer Society Press, Los Alamitos, California, USA.

[14] D. Binkley and M. Harman. Analysis and visualization of predicate dependence on formal parameters and global variables. *IEEE Transactions on Software Engineering*, 30(11), November 2004.

[15] D. Binkley and M. Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62, 2004.

[16] D. Binkley, M. Harman, and J. Krinke. Characterizing, explaining, and exploiting the approximate nature of static analysis through animation. In *Proc. of the IEEE Workshop on Source Code Analysis and Manipulation*, Philadelphia, USA, September 2006.

[17] A. Binstock. Extra-strength code cleaners, January 2006. www.infoworld.com/article/06/01/26/74270_05FEcodelint_1.html.

[18] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, New York, NY, USA, 2003. ACM Press.

[19] R. Brooks. *Towards a theory of comprehension of computer programs*, volume 18. ACM Press Journal Man-Machine Studies, 1983.

[20] G. Canfora, A. Cimitile, and M. Munro. RE^2: Reverse engineering and reuse re-engineering. *Journal of Software Maintenance : Research and Practice*, 6(2), 1994.

[21] G. Canfora and M. Di Penta. New frontiers of reverse engineering. *Future of Software Engineering, L. Briand and A. Wolf (eds.), IEEE-CS Press*, 2007.

[22] B. Carey. Accidental invention points to end of light bulbs, October 2005. www.livescience.com.

[23] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In ACM, editor, *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Charleston, South Carolina, January 10–13, 1993*, New York, NY, USA, 1993. ACM Press.

[24] H. Cleve and A. Zeller. Locating causes of program failures. In *Proc. of the 2005 International Conference on Software Engineering*, St. Louis, Missouri, May 2005.

[25] J. Cobleigh, L. Clarke, and L. Osterweil. Flavers: A finite state verification technique for software systems. *IBM Systems Journal – Software Testing and Verification*, 41(1), 2002.

[26] M. Collard. Addressing source code using srcml. In *IEEE International Workshop on Program Comprehension Working Session: Textual Views of Source Code to Support Comprehension (IWPC'05)*, 2005.

[27] J. Cordy, T. Dean, A. Malton, and K. Schneider. Source transformation in software engineering using the TXL transformation system. *Information and Software Technology*, 44(13), 2002.

[28] J. Cordy et. al. eds. *International Workshop on Program Comprehension*. Computer Society Press, St. Louis, MO, 2005.

[29] Coverity. www.coverity.com, 2006.

[30] *9th European Conference on Software Maintenance and Reengineering*, Manchester, UK, March 2005. IEEE Computer Society.

[31] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4), 1991.

[32] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*, Berlin, Germany, Mar. 1996. IEEE Computer Society Press, Los Alamitos, California, USA.

[33] B. Demsky, M. Ernst, P. Guo, S. McCamant, J. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *International Symposium on Software Testing and Analysis (ISSTA)*, July 2006.

[34] M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, and W. Visser. Formal software analysis : Emerging trends in software model checking. *Future of Software Engineering, L. Briand and A. Wolf (eds.), IEEE-CS Press*, 2007.

[35] Edison Design Group. Compiler front ends, 2006.

[36] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Notices*, 29(6), June 1994.

[37] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.

[38] M. Fahndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. *Proc. of the ACM SIGPLAN 98 Conference on Programming Language Design and Implementation*, 33(5), 1998.

[39] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3), July 1987.

[40] C. N. Fischer and R. J. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings Series in Computer Science. Benjamin/Cummings Publishing Company, Menlo Park, CA, 1988.

[41] R. France and B. Rumpe. Model-driven development of complex systems: A research roadmap. *Future of Software Engineering, L. Briand and A. Wolf (eds.), IEEE-CS Press*, 2007.

[42] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8), Aug. 1991.

[43] C. Gunter. Abstracting dependencies between software configuration items. *ACM Transactions on Software Engineering and Methodology*, 9(1), 2000.

[44] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. *Proc. of the Int. Conf. on Software Maintenance*, 1992.

[45] R. Gupta, M. Soffa, and J. Howard. Hybrid slicing: integrating dynamic information with static analysis. *ACM Transactions on Software Engineering and Methodology*, 6(4), 1997.

[46] S. Guyer and C. Lin. Client-driven pointer analysis. In *FCRC Static Analysis Symposium*, June 2003.

[47] B. Hackett and A. Aiken. How is aliasing used in systems software? In *SIGSOFT '06/FSE-14: Proc. of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA, 2006. ACM Press.

[48] M. Harman. The current state and future of search-based software engineering. *Future of Software Engineering, L. Briand and A. Wolf (eds.), IEEE-CS Press*, 2007.

[49] M. Harman, D. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1), Oct. 2003.

[50] M. Harman, C. Fox, R. M. Hierons, D. Binkley, and S. Danicic. Program simplification as a means of approximating undecidable propositions. In 7^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'99)*, Pittsburgh, Pennsylvania, USA, May 1999. IEEE Computer Society Press, Los Alamitos, California, USA.

[51] M. Harman, N. Gold, R. M. Hierons, and D. Binkley. Code extraction algorithms which unify slicing and concept assignment. In *IEEE Working Conference on Reverse Engineering (WCRE 2002)*, Richmond, Virginia, USA, Oct. 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[52] M. Harman, B. Korel, and P. Linos. Special issue on software maintenance and evolution. *IEEE Transactions on Software Engineering*, 31(10), 2005.

[53] M. Harrold. Testing: A roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[54] M. Heimdahl. Safety and software intensive systems: Challenges old and new. *Future of Software Engineering, L. Briand and A. Wolf (eds.), IEEE-CS Press*, 2007.

[55] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe. Automatic design pattern detection. In *IEEE International Workshop on Program Comprehension*, may 2003.

[56] R. M. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4), 1999.

[57] M. Hind. Pointer analysis — haven't we solved this problem yet? In *Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, Utah, USA, June 2001. ACM.

[58] S. Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Transactions on Programming Languages and Systems*, 19(1), 1997.

[59] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In 14^{th} *International Conference on Software Engineering*, Melbourne, Australia, 1992.

[60] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), 1990.

[61] V. Issarny, M. Caporuscio, and N. Georgantas. A perspective on the future of middleware-based software engineering. *Future of Software Engineering, L. Briand and A. Wolf (eds.), IEEE-CS Press*, 2007.

[62] D. Jackson and M. Rinard. Software analysis: A roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[63] M. Jazayeri. Web application development: The coming trends. *Future of Software Engineering, L. Briand and A. Wolf (eds.), IEEE-CS Press*, 2007.

[64] J. Jones and M. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[65] K. K7. www.klocwork.com, 2006.

[66] S. Kan. *Metrics and Models in Software Quality Engineering*. Addison Wesley, 2002.

[67] S. Khurshid and Y. Suen. Generalizing symbolic execution to library classes. In *6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, volume 31(1), New York, NY, USA, January 2005. ACM Press.

[68] P. Koopman. Elements of the self-healing system problem space. In *Workshop on Software Architectures for Dependable Systems (WADS)*, May 2003.

[69] B. Korel and J. Rilling. Dynamic program slicing in understanding of program execution. In 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)*, Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[70] A. Lakhotia and E. Kumar. Abstract stack graph to detect obfuscated calls in binaries. In 4^{th} *International Workshop on Source Code Analysis and Manipulation (SCAM 04)*, Chicago, Illinois, USA, Sept. 2004. IEEE Computer Society Press, Los Alamitos, California, USA.

[71] A. Lakhotia, J. Li, A. Wallenstein, and Y. Yang. Towards a clone detection benchmark suite and results archive. In *Proc. of the International Workshop on Program Comprehension*, Portland, Oregon, May 2003.

[72] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, FL, Jan. 1991. ACM Press.

[73] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92), SIGPLAN Notices*, July 1992. Published as SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92), SIGPLAN Notices, volume 27, number 7.

[74] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *14th International Conference on Program Comprehension*, 2006.

[75] D. Liang and M. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Proc. of the International Symposium on static Analysis*, 2001.

[76] C. Liu, L. Fei, X. Yan, J. Han, and S. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32(10), 2006.

[77] J. R. Lyle and M. Weiser. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications*, Peking, 1987. IEEE Computer Society Press, Los Alamitos, California, USA.

[78] M. Lyu. Software reliability engineering: A roadmap. *Future of Software Engineering, L. Briand and A. Wolf (eds.), IEEE-CS Press*, 2007.

[79] S. Mancoridis, B. S. Mitchell, Y.-F. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings; IEEE International Conference on Software Maintenance*. IEEE Computer Society Press, 1999.

[80] A. Marcus and J. Maletic. Identification of high-level concept clones in source code. In *Proc. of Automated Software Engineering*, San Diego, CA, November 2001.

[81] M. Martin and B. L. M. Lam. Finding application errors and security flaws using pql: a program query language. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, October 2005.

[82] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*,

14(2), 2004.

[83] A. Memon and Q. Xie. Studying the fault-detection effectiveness of gui test cases for rapidly evolving software. *IEEE Transactions on Software Engineering*, 31(10), 2005.

[84] M. Mendonca and N. Sunderhaft. Mining software engineering data: A survey, 1999.

[85] D. L. Métayer. Program analysis for software engineering: new applications, new requirements, new tools. *ACM Computing Surveys*, 28(4es), 1996.

[86] B. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3), March 2006.

[87] L. Moonen. Generating robust parsers using island grammars. In *Working Conference on Reverse Engineering*, 2001.

[88] G. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 1965.

[89] R. Muth and S. Debray. On the complexity of flow-sensitive dataflow analysis. In *Proc. of the 27th Annual Symposium on Principles of Programming Languages*, 2000.

[90] H. Nilsson and P. Fritzson. Lazy algorithmic debugging: Ideas for practical implementation. In P. A. Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*. Springer Verlag, May 1993.

[91] J. Ning, A. Engberts, and V. Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5), 1994.

[92] S. Pheng and C. Verbrugge. Dynamic data structure analysis for Java programs. In *ICPC '06: Proc. of the 14th IEEE International Conference on Program Comprehension*. IEEE Computer Society, 2006.

[93] H. Pohlheim. Visualization of evolutionary algorithms - set of standard techniques and multidimensional visualization. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[94] A. Pretschner, M. Broy, I. Krüger, and T. Stauner. Software engineering for automotive systems: A roadmap. *Future of Software Engineering, L. Briand and A. Wolf (eds.), IEEE-CS Press*, 2007.

[95] F. Qian and L. Hendren. Towards dynamic interprocedural analysis in jvms. In *Proc. of the 3rd Virtual Machine Research and Technology Symposium*, San Jose, USA, May 2004. Usenix.

[96] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(1), 2000.

[97] D. Ratiu and F. Deißenböck. How programs represent reality (and how they don't). In *Proc. of the 13th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, 2006.

[98] T. Reps. Program analysis via graph reachability. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40(11 and 12). Elsevier Science B. V., 1998.

[99] J. Rilling and S. P. Mudur. On the use of metaballs to visually map source code structures and analysis results onto 3d space. In 10th *Working Conference on Reverse Engineering*, Richmond, Virginia, Oct. 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[100] E. Ruf. Context-insensitive alias analysis reconsidered. *ACM SIGPLAN Notices*, 30(6), June 1995.

[101] S. Rugaber. *Program Comprehension*, volume 35(20). Marcel Dekker, Inc. New York, 1995.

[102] B. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proc. of the Twelfth International Conference on Compiler Construction*, Warsaw, Poland, April 2003.

[103] D. Saff and M. Ernst. Reducing wasted development time via continuous testing. In *Fourteenth International Symposium on Software Reliability Engineering*, Denver, CO, November 17–20, 2003.

[104] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3), 2002.

[105] K. Sartipi. *Software Architecture Recovery based-on Pattern Matching*. PhD thesis, University of Waterloo, School of Computer Science, 2003.

[106] D. Schmidt. Structure-preserving binary relations for program abstraction. *Essence of Computation: Complexity, Analysis, Transformation*, 2002.

[107] G. Schulmeyer and J. Mcmanus. *The Handbook of Software Quality Assurance (3rd Edition)*. Prentice-Hall, Inc., 1999.

[108] N. Shahmehri. *Generalized algorithmic debugging*. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1991. Available as Linköping Studies in Science and Technology, Dissertations, Number 260.

[109] J. Silva. Algorithmic debugging strategies. In *International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2006)*, 2006.

[110] Sjøberg, T. Dybå, and M. Jørgensen. The future of empirical methods in software engineering research. *Future of Software Engineering, L. Briand and A. Wolf (eds.), IEEE-CS Press*, 2007.

[111] D. Strein, H. Kratz, and W. Löwe. Cross-language program analysis. In *Proc. of 2006 IEEE Workshop on Source Code Analysis and Manipulation (SCAM'06)*, Philadelphia, USA, September 2006.

[112] A. Stuckenholz. *Component evolution and versioning state of the art*. ACM Press, 2005.

[113] A. van Lamsewwrde. Formal specification: A roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[114] D. Weise, R. F. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: Representation without taxation. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM SIGACT and SIGPLAN, ACM Press, 1994.

[115] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4), 1984.

[116] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. *ACM SIGPLAN Notices*, 30(6), June 1995.

[117] M. Woodside, G. Franks, and D. Petriu. The future of software performance engineering. *Future of Software Engineering, L. Briand and A. Wolf (eds.), IEEE-CS Press*, 2007.

[118] H. Yang and M. Ward. *Successful Evolution of Software Systems*. Artech House, 2003.

[119] A. Zeller. The future of programming environments. *Future of Software Engineering, L. Briand and A. Wolf (eds.), IEEE-CS Press*, 2007.