Representing Programs in Multiparadigm Software Development Environments*

Scott Meyers and Steven P. Reiss

Department of Computer Science
Brown University, Box 1910
Providence, RI 02912

Abstract

In this paper, we describe a canonical program representation, Semantic Program Graphs (SPGs), and we show how SPGs can act as the foundation for multiparadigm software development environments. Using SPGs as the basis for program representation allows developers to see different views of programs that correspond to different ways of thinking about them, and it allows editors to be created so that the underlying program may be edited using any of the paradigms. As the sole program representation, SPGs also facilitate communication between paradigms: changes made in one view can be immediately reflected in all other views.

Introduction

During the design, implementation, maintenance, and documentation of software systems, programmers tend to think about their jobs in different ways, depending on the task currently facing them. Each of these different ways of thinking reflects a different *paradigm* of programming, and each paradigm may be visualized via one or more *views* of a software system. Some examples of this kind of *multiparadigm programming* are:

- During system design, software developers often use high-level dataflow paradigms such as SADT[27] or SREM[3]. Views in these paradigms are typically graphical, using boxes and arrows to represent data and activities, plus text to annotate the graphs.
- When dealing with parallel programs consisting of communicating tasks, they may use a paradigm based on Petri Nets[19]. This paradigm is also graphical, using circles, lines and arrows to represent places, transitions, and inputs/outputs.
- They may write part or all of their programs using a specific language paradigm, such as FORTRAN, Ada, or Lisp. Views of these languages are usually textual, although graphical views such as Nassi-Schneiderman diagrams[16] or flowcharts are sometimes used.
- During testing, they may assess the adequacy of their test data using a structural coverage paradigm, i.e., branch coverage or statement coverage[17]. Views in this paradigm could be either textual (e.g., a listing showing the percentage coverage for each module name) or graphical (e.g., a picture of the paths through a module's control flow graph, with the covered branches and/or statements highlighted).
- At the outset of a new maintenance task on an unfamiliar program, they may approach the understanding of a program using a paradigm that suppresses all "irrelevant" parts of a program. Possible views for such a paradigm are program slices[30, 13] and local dataflow behavior (i.e., def-use chains).
- For documentation purposes, they may employ a presentation paradigm emphasizing a top-down hierarchical description of the system. A useful view in such a paradigm would be a graphical depiction of the system's call graph.
- During system execution, programmers may visualize their system's dynamic behavior using either a control flow paradigm or a dataflow paradigm.

*Support for this research was provided by the NSF under grant DCR 8605567; by DARPA under contract N00014-83-K-0146, ARPA order 6320; and by the Digital Equipment Corporation under agreement 393.

0730-3157/89/0000/0420$01.00 © 1989 IEEE

A control flow view could show which statement(s) were being executed or were eligible for execution, while a dataflow view could show which data definitions were flowing to particular uses.

In fact, the number of possible paradigms and views is limited only by the imagination of programmers, and research into software development environments is beginning to explore environments that allow programmers to define their own views.

One such multiparadigm environment is Garden[22, 21], developed by Steven P. Reiss at Brown University. Within Garden, programmers create new textual and/or graphical programming languages (views) by defining the syntax and semantics of the new view in terms of views that are either built into Garden or that have already been defined within it. A single program can then be created and modifed using any number of views simultaneously. Unfortunately, the view creator must write a two-way translator for each pair of views that are to be kept consistent; this is a daunting task when the number of views is large. A preferable arrangement would be to have a single canonical representation of program semantics within Garden. Then adding a new view to the environment would require the creation of only one two-way mapping: from the view to the canonical representation and from the representation to the view; consistency maintenence between views could be handled by the representation itself. In this paper, we describe a representation for program semantics that could serve as a canonical representation for programs in environments such as Garden.

Related Work

The utility of multiple views has been noted by researchers in a number of fields. Generalized user interface management systems such as Segue[28] and Higgens[12] offer flexible representations for applications' data structures based on trees (Segue) or graphs (Higgens). These representations could conceivably support a large number of views, but research on these systems has focused on interface management, and has not addressed the problem of how programs might be effectively modeled. Similarly, Alpern and his collegues have developed efficient methods for incrementally evaluating attributed graphs in a software development environment[4], but they have not proposed specific program representations. Rich and Waters suggest that the language-independent Plan Calculus program representation employed in the Programmer's Apprentice project[25] would be well-suited to supporting multiple views, but they have so far chosen not to pursue this avenue of research[26].

Most work on multiple program views has taken place in the context of tightly integrated software development environments, and the most common internal representation for programs in such environments is the abstract syntax tree (AST). ASTs are at the core of many well-known programming environments, including the Cornell Program Synthesizer[24], Gandalf[9], Pecan[20], R^n[11], MENTOR[7], and CENTAUR[5]. The centerpiece of these environments is a language-based editor, which usually offers incremental syntactic and semantic analysis of the program being edited, plus incremental code generation. An AST is an excellent representation for these syntax-based views. Experience with Pecan has shown, however, that ASTs are too restrictive in the range of views that they can conveniently represent[21]. ASTs are not well suited for views that are unrelated to the syntax of a textual language, and are in general not flexible enough for programmers who wish to define arbitrary new views during program development. In fact, none of these environments allows programmers to define new views. In addition, their one-dimensional nature makes them particularly ill-suited for two- or three-dimensional graphical views. Similar difficulties arise with program representations based on abstract syntax graphs, such as those used in IPSEN[15] and TEAM[6].

A representation for programs that is gaining increasing attention is the program dependence graph(PDG) [8]. A PDG identifies those portions of a program that are truly control- or data-dependent on other parts of the program, and it directly represents those dependencies. Because it isolates such dependencies, the PDG has found favor amongst researchers who wish to automatically produce parallel code from programs written in serial languages[1], but it has also been suggested that a PDG would be a good program representation for a software development environment[18]. Unfortunately, the form of a PDG is very different from the views that programmers typically use (i.e., control flow, data flow, etc.), and it would be an arduous task to write translators that would map between views and a PDG. For an environment in which defining new views is a common operation, this is a serious drawback. In addition, PDGs so far lack efficient incremen-

[1]Sequential languages typically overspecify statement sequences, therefore statements that are independent appear in a program as if they were dependent on one another. For example, if statement S_1 immediately precedes statement S_2 in an Algol-like language, there will appear to be a control dependency between S_1 and S_2, even if S_1 and S_2 deal with entirely different sets of variables. A PDG would show whether such statements were independent. If they were, they would be eligible for execution in parallel.

tal algorithms for common operations in an interactive environment, such as adding a statement to a program under construction.

Finally, as part of his work on a General Design Representation[14], Lubars has proposed a view-independent representation for design information. However, that research has focused on static design data, and has not addressed issues concerning the executability of the resulting representation. Because programs are fundamentally executable entities, we believe it is crucial that our representation be executable.

Requirements

The foremost requirement of a semantic representation is that it provide a large enough collection of semantic primitives so that a broad class of views[2] can be directly expressed within the representation. In addition, these primitives must be at roughly the same level of abstraction as are those of the class of views to be supported – otherwise it becomes too difficult to write translators between the views and the representation. Informally, we say that the representation must accurately reflect "what is going on" in a program, because when translating between views, we want to be able to preserve the "spirit" of a program.

For example, if a program with a loop in it is created using a view that lacks structured looping constructs and instead uses unstructured GOTOs, we would still like the semantic representation of that program to reflect the fact that a loop is present. That is, "what is going on" in the program is a loop, even if the concrete syntax of a view can't express that directly. Hence our representation must support the notion of a loop as a semantic primitive.[3]

We believe that in order to accurately characterize "what is going on" in a program, a semantic representation must support at least the following aspects of programming:

- **Different language paradigms:** Imperative languages (e.g., FORTRAN, C, Algol-like languages), functional languages (e.g., Lisp, Miranda), and dataflow languages (e.g., Val, Id).
- **Serial and parallel constructs**, including dynamic process creation and synchronous and asynchronous interprocess communication.
- **Deterministic and nondeterministic constructs**, including nondeterministic branching and acceptance of interprocess communication (e.g., Ada's SELECT statement).
- **Control flow**, including looping, branching, jumping (GOTOs), and subprogram calls.
- **Dataflow** within a program, either as directly expressed in a dataflow-based design language or a dataflow-oriented programming language, or as derived from a control flow graph.
- **Name visibility**, including scopes, "private" identifiers, and explicit name exportation and importation.

In addition to supporting these basic concepts, we also require that our semantic representation be directly executable, because we want to support dynamic views as well as static views. We further require that it be modifiable using efficient incremental algorithms, since the kinds of multiparadigm development environments we wish to support are highly interactive.

Semantic Program Graphs

The core of our representation is a type of program flow graph called a *Semantic Program Graph* (SPG). The nodes of the graph represent operations to be performed, and the arcs correspond to control flow and/or data flow. An SPG includes aspects of conventional control flow graphs and also of the "machine language" graphs of dataflow programming languages[1]. It was originally inspired by the combined model of computation developed by Treleaven and his colleagues[29], but we have extended their model to include hyperarcs, arbitrary graph annotations, and "groupings" (arbitrary sets of nodes and/or arcs).

Structure

Figure 1 shows the basic building block of an SPG. The nodes O_1 and O_2 represent operations to be performed, P represents a controlling predicate, and T represents

[2] We do not claim to offer a semantic representation that will suffice for all possible views. Instead, we claim to offer a representation that will support a broad class of views that are commonly useful. In this sense our goal is less ambitious than, say, that of denotational semantics or other formal semantic models. On the other hand, we *do* aim to provide a representation that will facilitate the mapping from semantics back to the concrete syntax of a number of languages, an undertaking which is not attempted by formal semantic models.

[3] For such an unstructured language, the responsibility for recognizing the presence of a loop implemented using GOTOs rests with the writer of the view-to-semantics translator, not with the semantic representation itself. That is, our representation does not attempt to "discover" loops in unstructured programs. However, it *is* responsible for providing a way to represent the loop *as a loop*, since other views are likely to be able to express the loop directly.

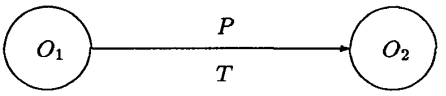


Figure 1: Building Block of an SPG

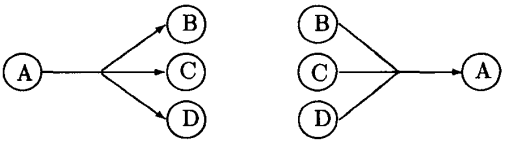
a token that flows from O_1 to O_2 when P is true immediately after executing O_1. If we interpret this graph as showing control flow, then it corresponds to the idea, "Perform O_1. If P, then perform O_2." If we interpret it as showing data flow, then it corresponds to the idea, "Perform O_1, producing data value T. If P, then perform O_2 using T." In general, nodes represent operations to be performed, while arcs and their associated predicates control which nodes are executed.

During program execution, tokens are created at nodes that are executed. Such a node places a single token on each of its outarcs. The tokens then flow along the arcs to other nodes. A node is available for execution whenever it has a token on all of its inarcs. In a standard sequential language, each node has a single inarc and a single outarc, and during execution a single token moves around the graph. For parallel languages, many tokens may move around the graph at once, and many nodes may be available for execution at any given time.

Nodes

Because execution is controlled by arc predicates, there are relatively few primitive operations to be performed at nodes. Common operations are system primitives, and include assignment, computation of simple expressions, subprogram call, dynamic process creation, and I/O. These are the operations that the system "knows" about, and they are typically easy to translate from view to view.

Not all views have base operations that are easy to express using SPG primitives, however. Consider unification, which is a primitive in Prolog, but would have to be represented as a complex set of operations in an SPG. To handle such situations, our representation offers a complete programming language, and an SPG node can be given any amount of code in this language as its operation to be performed. This is an important feature of our representation: it guarantees that an SPG can be built to represent the semantics of *any* program. For views with primitives at roughly the same level of abstraction as SPG primitives, their semantics can be expressed directly. On the other hand, for views with primitives that differ substantially from SPG primitives, it is still possible to express the desired semantics in an SPG, the program will still be executable, and such views are still integrated with all other views sharing an SPG, as described later in this paper.



(a) (b)

Figure 2: Hyperarcs in an SPG

Arcs

In representing the semantics of programs, it is convenient to use hyperarcs instead of simple arcs between nodes. For example, the graph in figure 2a can be used to represent any of the following control flow concepts:

- **Multiway Branch**: after executing A, execute one of B, C, or D, depending on the program state;
- **Nondeterministic Choice**: after executing A, randomly choose one of B, C, or D to execute;
- **Parallel Branch**: after executing A, execute each of B, C, and D in parallel.

Figure 2a can also be given a dataflow interpretation: definition A can reach the uses B, C, and D.

Similarly, figure 2b has a number of useful interpretations. In a control flow context, it could mean that the immediate predecessor of A must be one of B, C, and D. In a dataflow context, it could mean that the use at A must have been defined at B, C, or D.

Tokens

Strictly speaking, all tokens carry data values, but in practice it is convenient to think of tokens as being of four types. Control tokens carry meaningless values (the presence of the token itself is what is important), data tokens carry standard data values, reference tokens carry memory addresses, and graph tokens carry "pointers" to other SPGs. In a traditional control flow graph, all tokens are control tokens. In a traditional dataflow programming language graph, all tokens are data tokens. In a traditional control flow graph augmented with static dataflow information, reference tokens flow along arcs from definitions to uses. Data and

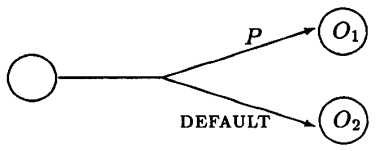


Figure 3: Representing an if/then/else Statement

reference tokens also correspond to by-value and by-reference parameter passing conventions, respectively. Graph tokens are used to dynamically create the structure of the SPG, and are useful for representing subprogram parameters, higher-order functions, and dynamically established interprocess communication.

Predicates

Arc predicates determine whether tokens are allowed to flow along particular arc branches. They may be simple predicates that use only primitive operators (such as the relational operators), or they may be complex predicates with their own SPG representation at a lower level of abstraction (e.g., boolean functions). There is also the predefined predicate DEFAULT, which evaluates to true iff all other branches of a hyperarc evaluate to false. It is useful for representing the "else" clause of if/then/else constructs, as well as the "default" clause of multiway branches (e.g., C's SWITCH statement). Figure 3 represents the concept, "if P then O_1 else O_2."

Annotations

The structure of an SPG faithfully reflects the execution semantics of the underlying program, but it is not always adequate for representing the less precise notion of "what is going on." For example, if we notice that a particular variable is only used in one region of a program, it might be because that variable is private to that part of the program, or it might simply be happenstance. Semantically, these two situations are very different, but the graph itself offers us no guidance in distinguishing them. To overcome this kind of problem, our representation uses graph annotations to add semantic information to the graph that is not readily available from its structure.

Annotations are arbitrary *(name, value)* pairs that may be attached to objects in the graph, including nodes, arcs, annotations, and groupings (see below). Annotations are much like Unix™ environment variables or X Window System™ options: a few of them have special significance to the system, but most of them are meaningful to only one or a few programs. Individual views might use them to keep track of formatting information for the concrete syntax of the view, to store dynamic information such as profiling or test coverage data, to record comments, etc. Annotations are very flexible.

System-defined annotations are primarily concerned with capturing the "flavor" of a program (such as distinguishing structured loops from incidental GOTOs) and with representing a program's non-structural semantic features (such as rules controlling name visibility). Among the most important annotations understood by the SPG are those used to

- **Identify the "start" node of a program.** Each program begins execution at a particular node in the SPG; an annotation is used to identify this node.
- **Identify structured loops in a program.**
- **Store statement labels.** Some views may use statement labels, some may not. If a program was assigned labels during creation, those must be saved, even though other views might not display them. If a program was created without labels, some views may need to assign them in order to accurately display the underlying graph structure of the program.
- **Demarcate scopes.** Data encapsulation is typically achieved by controlling name visibility in a program. Scopes are the means for this control. Scopes do not correspond to individual graph components, but to portions of the graph. Such graph partitioning is achieved using groupings, and annotated groupings are the basis for scopes in an SPG. Groupings and scopes are discussed in detail below.

Groupings

Often it is convenient to think of a part of an SPG as a single semantic entity. Useful parts may include subgraphs, sets of nodes, or sets of arcs. Such partitions are established in SPGs using *groupings*. For example, one might be interested in a subgraph corresponding to a particular subprogram or process; in the grouping corresponding to the set of nodes that used more than 5% of the CPU time on the last program run; or in the grouping for the set of control arcs not covered by a particular test set. Because groupings may be completely arbitrary, they are very flexible.

Groupings may be annotated. Like structural annotations, grouping annotations need not have any *a*

priori meaning, but one of the most important grouping annotations is predefined: the *scope* annotation. Scopes are used to demarcate name spaces in a program. They do this by defining rules for

- **name importing**: the conditions under which names defined outside the scope are visible inside the scope;
- **name exporting**: the conditions under which names defined inside the scope are visible outside the scope;
- **name conflicts**: what is done when a visible name is defined in the current scope as well as in another scope;
- **name lookup**: how the scope of a name is determined when the name is not defined in the current scope.

Scope rules in programming languages are remarkably varied. For example, name importation and exportation can be done by explicit mention of the names involved, by using an Algol-style "lexically closest block" rule, by importing and exporting all names, or by importing and exporting no names at all[4]. Similarly, name conflicts can be dealt with by having local names prevail or by banning name conflicts altogether. Name lookup can be performed in the Algol-style or by requiring that all names be uniquely scoped. In our representation, we assign predefined meanings to certain scope annotations, and view writers use these annotations to achieve the kind of name visibility they desire. Our predefined annotations are general enough to describe a wide variety of scope rules, including all those discussed in this section.

Maintaining Multiple Views

An interactive multiparadigm software development environment is a highly dynamic entity. A single program may simultaneously be edited by several people, each of whom employs several different views[5]. Our architecture for organizing the interactions between the various views and the SPG is shown in figure 4. Each

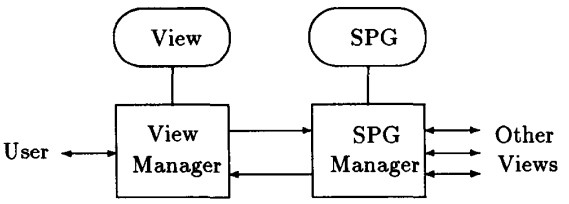


Figure 4: Architecture of View-SPG Interaction

instance of a view is controlled by a view manager, which is responsible for interacting with the user and controlling the physical appearance of its view. When modifications to the view call for modifications to the underlying SPG, the view manager communicates the changes to the SPG manager, which modifies the SPG. The SPG manager then broadcasts the changes in the SPG to all views, including the view that originally initiated the change[6]. Each view manager then updates its view in accord with the changes made to the SPG.

Three points are relevant here. First, some changes to a view may be semantically cosmetic; these changes are handled entirely by the view manager. Second, the changes broadcast to the views by the SPG manager may be more extensive than the original modification received by the SPG manager, because the SPG manager may perform incremental anaysis on the objects in the SPG. For example, a view might tell the SPG manager to remove a node from the SPG, but that might result in modifications to the dataflow arcs in the SPG also being made. Third our architecture has view managers treat all changes to the SPG uniformly – local modifications are dealt with in exactly the same manner as are remote modifications.

The SPG is designed to take advantage of efficient incremental algorithms for computing the effects of control flow changes on dataflow, and vice versa. Algorithms are now known for computing dataflow information incrementally from control flow graphs[31], and computing control flow information from dataflow graphs is a simple matter of performing a topological sort on graph nodes using dataflow arcs as the ordering criterion. (However, computing sequential control flow information from a dataflow graph is unlikely to be a common operation, since it reduces an inherently parallel program to a sequential program.)

[4] Scope rules can be used to achieve the effects of name "inheritance" without having a special rule for inheritance itself. If scope C is nested inside both scopes A and B, then using an Algol-style "nested scope" rule allows C to "multiply inherit" names from both A and B, even if A and B are not nested within one another.

[5] Concurrent access by many agents to a single object brings up all the familiar access-control issues that pervade database management. In this paper, we ignore such issues, because they are primarily germane to the *access* to the program representation, not the representation itself.

[6] An alternative approach is to use *selective broadcasting*, such as is used in FIELD[23]. With selective broadcasting, the SPG manager would only broadcast changes to views that had previously registered an interest in the type of change being communicated.

Status

We have successfully performed "paper designs" of SPGs as a semantic representation for a number of languages, including deterministic and nondeterministic FSAs, Petri Nets, high-level dataflow diagrams (e.g., SADT), and simplified Pascal. We have also used SPGs to represent task communication in Ada, and to represent a form of the inheritance characteristic of object-oriented languages. We plan to continue our experiments by working on representations for dataflow graphs, a Linda-class language[2], and Higraphs[10]. In addition, we are looking at how to add constructs for demand-based (lazy) evaluation, and at how to model data structures, including variables allocated statically, on a stack, and from the heap. Finally, we are beginning work on a prototype implementation of SPGs and an SPG manager that will allow us to gain practical experience with the representation. We plan to eventually incorporate our work on SPGs into the tightly integrated Garden environment, and possibly also into the more loosely integrated FIELD environment, both of which are under development at Brown University.

References

[1] William B. Ackerman. Data Flow Languages. *IEEE Computer*, pages 15 – 24, February 1982.

[2] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and Friends. *IEEE Computer*, pages 26 – 34, August 1986.

[3] Mark Alford. SREM at the Age of Eight: The Distributed Computing Design System. *IEEE Computer*, 18(4):36 – 46, April 1985.

[4] Bowen Alpern, Alan Carle, Barry Rosen, Peter Sweeney, and Kenneth Zadeck. Graph Attribution as a Specification Paradigm. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 121–129. ACM Press, November 1988. Published as *ACM Software Engineering Notes* 13:5, November 1988, and *ACM SIGPLAN Notices* 24:2, February 1989.

[5] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi an G. Kahn, B. Lang, and V. Pascual. CENTAUR: the System. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24. ACM Press, November 1988. Published as *ACM Software Engineering Notes* 13:5, November 1988, and *ACM SIGPLAN Notices* 24:2, February 1989.

[6] Lori A. Clarke, Debra J. Richardson, and Steven J. Zeil. TEAM: A Support Environment for Testing, Evaluation, and Analysis. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 153–162. ACM Press, November 1988. Published as *ACM Software Engineering Notes* 13:5, November 1988, and *ACM SIGPLAN Notices* 24:2, February 1989.

[7] Véronique Donzeau-Gouge, Gérard Huet, Gilles Kahn, and Bernard Lang. Programming Environments Based on Structured Editors: The MENTOR Experience. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*, chapter 7, pages 128 – 140. McGraw-Hill, New York, 1984.

[8] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319 – 349, July 1987.

[9] Nico A. Habermann and David Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, December 1986.

[10] David Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514 – 530, May 1988.

[11] Robert T. Hood and Ken Kennedy. A Programming Environment for FORTRAN. Technical Report TR84-1, Rice University, June 1984.

[12] Scott E. Hudson and Roger King. Semantic Feedback in the Higgens UIMS. *IEEE Transactions on Software Engineering*, 14(8):1188–1206, August 1988.

[13] Hareton K. N. Leung and Hassan K. Reghbati. Comments on Program Slicing. *IEEE Transactions on Software Engineering*, SE-13(12):1370–1371, December 1987.

[14] Mitchell D. Lubars. A General Design Representation. Technical Report STP-066-89, Microelectronics and Computer Technology Corporation, February 1989.

[15] Manfred Nagl. A Software Development Environment Based on Graph Technology. Technical Report 87-3, Lehrstuhl für Informatik III, Rheinisch-Westfälische Technische Hochschule

Aachen, Ahornstraße 55, 5100 Aachen, West Germany, 1987.

[16] I. Nassi and B. Schneiderman. Flowchart Techniques for Structured Programming. *SIGPLAN Notices*, 8(8), August 1973.

[17] Simeon C. Ntafos. A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, 14(6):868 – 874, June 1988.

[18] Karl J. Ottenstein and Linda M. Ottenstein. The Program Dependence Graph in a Software Development Environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177 – 184, 1984. Also published as the May 1984 issue of *SIGPLAN Notices* and the May 1984 issue of *Software Engineering Notes*.

[19] James L. Peterson. Petri Nets. *Computing Surveys*, 9(3):223–252, September 1977.

[20] Steven P. Reiss. PECAN: Program Development Systems that Support Multiple Views. *IEEE Transactions on Software Engineering*, March 1985.

[21] Steven P. Reiss. GARDEN Tools: Support for Graphical Programming. Technical Report CS-86-12, Brown University, Department of Computer Science, April 1986.

[22] Steven P. Reiss. Working in the Garden Environment for Conceptual Programming. *IEEE Software*, pages 16 – 27, November 1987.

[23] Steven P. Reiss. Integration Mechanisms in the FIELD Environment. Technical Report CS-88-18, Brown University Computer Science Department, 1988.

[24] Thomas Reps and Tim Teitelbaum. The Synthesizer Generator. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, May 1984.

[25] Charles Rich. A Formal Representation for Plans in the Programmer's Apprentice. In *Proceedings of the Seventh international Joint Conference on Artificial Intelligence*, pages 1044–1052, August 1981. (IJCAI-81).

[26] Charles Rich and Richard C. Waters. The Programmer's Apprentice: A Research Overview. *IEEE Computer*, pages 10–25, November 1988.

[27] Douglas T. Ross. Applications and Extensions of SADT. *IEEE Computer*, pages 25 – 34, April 1985.

[28] Stuart C. Schaffner and Martha Borkan. Segue: Support for Distributed Graphical Interfaces. *IEEE Computer*, pages 42–55, December 1988.

[29] Philip C. Treleaven, Richard P. Hopkins, and Paul W. Rautenbach. Combining Data Flow and Control Flow Programming. *The Computer Journal*, 25(2):207 – 217, 1982.

[30] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984. Important corrections to this article can be found in [13].

[31] Frank Kenneth Zadeck. Incremental Data Flow Analysis in a Structure Program Editor. In *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction*, pages 132–143, June 1984.