A System for Multiparadigm Development of Software Systems

Scott Meyers
Steven P. Reiss

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-91-50
August 1991

Submitted to the Sixth International Workshop on Software Specification and Design, Como, Italy, October 25-2 1991.

A System for Multiparadigm Development of Software Systems

Scott Meyers and Steven P. Reiss

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

Abstract

It is our goal to create a software development environment that will offer multiple editable views of software systems, will allow users to define new views, and will support communication between views as the systems are changed. Our research to date has focused on the development of a single canonical representation for software systems, Semantic Program Graphs (SPGs). In this paper, we briefly describe SPGs, show how they can form the basis of a development environment, give examples of their utility, and compare them with other ways of representing software systems. We concentrate on describing the various characteristics of software systems that SPGs must support, and on explaining how SPGs can serve as the basis for the development environment we envision.

Introduction

During the process of specification, design, implementation, maintenance, and documentation of software systems, developers tend to think about their jobs in different ways, depending on the task currently facing them. Each way of thinking reflects a different *paradigm* of programming, and each paradigm may be visualized through one or more *views* of a software system. Common views include high-level data flow diagrams, Petri Nets, Finite State Machines, source code, test coverage data, program slices, def-use chains, call graphs, and execution. In theory, the number of possible paradigms and views is limited only by the imagination of programmers. Ideally, a software development environment should support this natural plethora of views, should allow developers to create new views without undue difficulty, and should automatically maintain consistency between disparate views of the same system, even while the views are being modified.

Unfortunately, current development environments fall short of this ideal, either because they offer only a single view, they fail to allow the creation of new views, or they fail to support communication between views. For example, a number of "multiple-view" environments are based on abstract syntax trees, but they offer only a restricted set of predefined views. The PRISMA system [1] is explicitly designed to offer multiple views of software specifications, and the views communicate with one another, but the views are fixed in advance. In addition, PRISMA suffers from the drawback that a two-way translator must be written for each pair of views that is to be kept consistent, so even if a provision for adding new views were provided, it would become unreasonably onerous to define new views as the number of views increased. This is the same problem suffered by the Garden environment [2], perhaps the only system specifically developed to encourage the creation of new views.

A preferable arrangement would be to have a single canonical representation for software semantics at the core of a development environment. 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 maintenance between views could be handled by the representation itself. In this paper, we briefly describe such a canonical representation, the SPG, and we show how it can be used as the basis for a development environment. Details on SPGs can be found elsewhere [3, 4, 5].

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 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, one wants to be able to preserve the "spirit" of a program.

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

- **Different control paradigms:** Imperative languages (e.g., FORTRAN, C, Algol-like languages), functional languages (e.g., Lisp, Miranda), and data flow languages (e.g., Id, Lucid).
- **Both serial and parallel constructs**, including dynamic process creation and synchronous and asynchronous interprocess communication.
- **Both deterministic and nondeterministic constructs**, including nondeterministic branching and acceptance of interprocess communication (e.g., Ada's select statement).
- **Both strict and non-strict parameter passing**, where a routine R is non-strict with respect to a parameter p if it is possible that $R(p) \neq \perp$ when $p = \perp$.
- **Control flow information**, including branching, jumping (gotos), looping, and routine calls.
- **Data Flow information** within a program, either as directly expressed in a data flow-based design language or a data flow-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 a semantic representation be directly executable, because we want to support dynamic views as well as static views.

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 edges correspond to control flow and/or data flow.[1] An SPG includes aspects of conventional control flow graphs and also of the "machine language" graphs of data flow programming languages. It was originally inspired by the combined model of computation developed by Treleaven and his colleagues[6], but we have extended their model to include hyperedges, arbitrary graph annotations, and "groupings" (arbitrary sets of nodes and/or edges). In our handling of routine calls, we have been influenced by the dataflow graphs produced for the Tagged-Token Dataflow Architecture [7] by Arvind and his coworkers. The description of SPGs presented here is necessarily abbreviated; full details are available elsewhere [5].

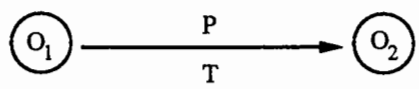

Figure 1: Building Block of an SPG

Figure 1 shows the basic conceptual 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 a token that flows from O_1 to O_2 when P is true immediately after executing O_1. In the context of control flow, it corresponds to the idea, "Perform O_1. If P, then perform O_2." In the context of data flow, 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 edges and their associated predicates control which nodes are executed.

Nodes

Because execution is controlled by edge predicates, there are relatively few primitive operations to be performed at nodes. Common operations include assignment, computation of simple expressions, routine call, 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

[1] SPGs are actually more general than this, because they incorporate some of the features found in hypertext systems. In particular, nodes may or may not be executable, and edges correspond to arbitrary relations. For example, parts of programs may have edges to and/or from "comment" nodes. This type of nonexecutable information is particularly important during specification and design, but in this paper, we restrict ourselves to the case where nodes and edges have a particular executable semantics.

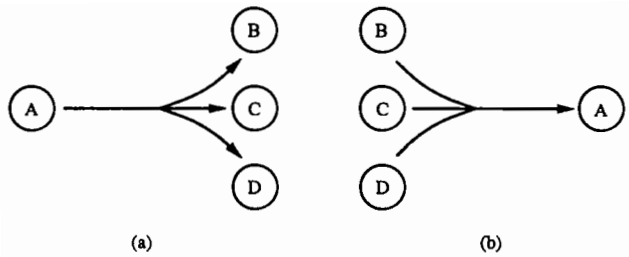

Figure 2: Hyperedges in an SPG

in an SPG. Rather than encode such complex primitives using base SPG operations, a view may add a "black box" node to an SPG. The execution semantics of such a node are to call a specified external routine whenever execution of the node is necessary. The semantics of the node will be invisible to most views, but the program will still be fully executable. This is an important feature of the representation: it guarantees that an SPG can be built to represent the execution semantics of *any* program.

Edges

In representing the semantics of programs, it is convenient to use hyperedges instead of simple edges 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 used in a data flow context: 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 data flow context, it could mean that the use at A must have been defined at B, C, or D.

The semantics of any particular edge is determined by the type of the edge. For example, one type of edge is used to represent conditionals (e.g., if/then/else and case constructs), while another type is used to represent nondeterministic and parallel execution.

Edges must originate at nodes, but they may terminate at nodes or edges. The use of edges terminating at edges is discussed below.

Tokens

Strictly speaking, all tokens carry data values, but in practice it is convenient to think of tokens as being of three types. Control tokens carry meaningless values (the presence of the token itself is what is important), data tokens carry standard data values, and routine tokens carry "pointers" to routines. In a traditional control flow graph, most tokens are control tokens. In a traditional data flow programming language graph, most tokens are data tokens. Routine tokens can be used to pass and/or return routines from other routines, thus lending direct support for views offering higher-order functions and function parameters.

Predicates

Edge predicates determine whether tokens are allowed to flow along particular edge 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 hyperedge 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).

Annotations

The structure of an SPG faithfully reflects the execution semantics and much of the static structure 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, edges, 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 presentation of the view,

to store dynamic information such as profiling or test coverage data, to record comments, etc.

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 edges. Such partitions are established in SPGs using *groupings*. For example, one might be interested in a subgraph corresponding to a particular routine or process. Groupings may, but need not, be nested within one another.

One of the most important types of groupings is the *scope* grouping. 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); and name lookup (how the scope of a name is determined when the name is not defined in the current scope).

Execution

The rules for executing SPGs are similar to those for FSAs and Petri Nets, but are more complex. In general, tokens are created at nodes that are executed. After execution, a node places a single token on each of its outgoing edges. The tokens then flow along the edges to other nodes and/or edges. A node or edge is eligible for execution whenever it has a token on all of its input edges. In a standard sequential language, each node has a single input edge and a single output edge, and during execution a single control token moves around the graph. For parallel languages, many control tokens may move around the graph at once, and many nodes may be eligible for execution at any given time.

It is worth emphasizing that each SPG component has a fixed, predefined semantics, and thus the composition and topology of a particular SPG determines the "meaning" of the software system being represented. Different views will display the SPG to users in different ways, but the semantics of the underlying SPG are explicitly independent of the views. It is this independence that allows SPGs to be executed by the SPG manager (see below) without worrying about which views are currently active.

Using SPGs

It is a theoretical impossibility to support mappings between all possible views; consider trying to map arbitrary programs into a language (view) with only two statements, HALT or LOOP FOREVER. In spite of this limitation, SPGs can be quite useful in practice. We have identified four broad classes of useful views that SPGs can support:

- **Control flow views.** These views are designed to directly provide information regarding the conditions under which control moves from one part of a system to another.
- **Data flow views.** These views are designed to directly provide information regarding how data flows through a system.
- **Execution views.** These views are designed to directly provide information regarding the current state of an executing system, often by dynamically "animating" a static view of the system.
- **Abstract views of system structure.** These views are designed to directly provide information regarding the overall structure of a software system, and as such usually abstract away details of the system in order to provide a more comprehensible presentation.

The first three of these classes correspond directly to the control flow edges, data flow edges, and executability of SPGs, respectively, and the fourth is closely related to SPG groupings, which tend to correspond to structural abstractions in software systems (e.g., functions, processes, etc.).

Table 1 lists a number of views that are often used with software systems, and shows how the information provided by those views relates to the categories just mentioned. A description of how each view could be implemented using SPGs can be found elsewhere [4].

Given these kinds of views, our architecture for organizing the interactions between the various views and the SPG is shown in figure 3. Each instance of a view is controlled by a view manager, which is responsible for interacting with the user and controlling the physical appearance (presentation) 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. Each view manager then updates its view in accord with the changes made to the SPG.

View	View Class			
	Control Flow	Data Flow	Execution	System Structure (Abstract)
Flowchart	√		√	
State Diagram	√		√	√
Petri Net	√		√	√
Call Graph	√		√	√
Statechart	√		√	√
Resource Profile			√	
Test Coverage			√	
Data Flow Diagram		√	√	√
Source Code	√		√	
Def-Use Chains		√	√	
Runtime Stack			√	
Build Dependencies		√		√

Table 1: Sample Views and the Types of Information they Provide

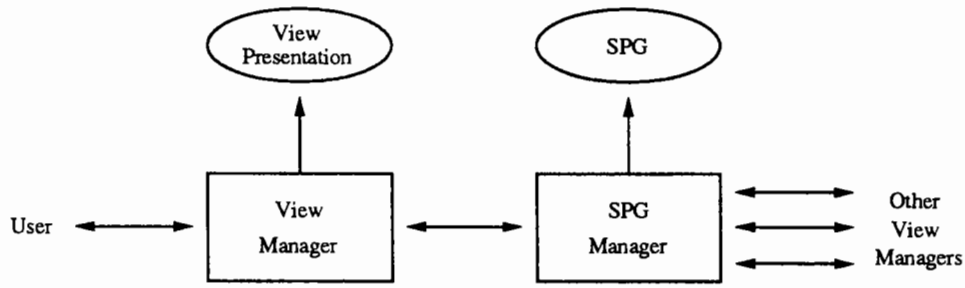

Figure 3: Architecture of View-SPG Interaction

Communication between view managers and the SPG manager is made possible by a set of predefined functions each offers for use by the other. For example, view managers offer functions to be called by the SPG manager when nodes begin evaluation, when tokens are moved along edges, when groupings are added or deleted, etc. Similarly, the SPG manager offers functions to be called by view managers whenever semantically meaningful changes are made to views, e.g., statements are added or deleted, etc. This approach to communication between representation and views is similar in spirit to the use of action routines in Gandalf [8].

Three points are relevant here. First, some changes to a view may be semantically cosmetic, e.g., the reformatting of a view's presentation. These changes are handled entirely by the view manager and do not affect the SPG or other views. 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 analysis 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 data flow edges in the SPG also being made. Third, this 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.

An Example

Consider the Petri Net shown in Figure 4. There are in some sense three "activities" depicted in this net: the P_1-P_2 cycle, the P_4-P_5 cycle, and the P_6-P_7 cycle. The P_1-P_2 cycle runs continuously, while control switches back and forth nondeterministically between the P_4-P_5 cycle and the P_6-P_7 cycle. At any given time, exactly one of these latter two cycles is running in conjunction with the P_1-P_2 cycle. We are interested in viewing this same system as a statechart.

Straightforward mappings based on local structural features of the Petri Net yields the SPG in Figure 5.[2] In the figure, SPG nodes are annotated with "place"

[2]These (quite simple) mappings are described elsewhere [4].

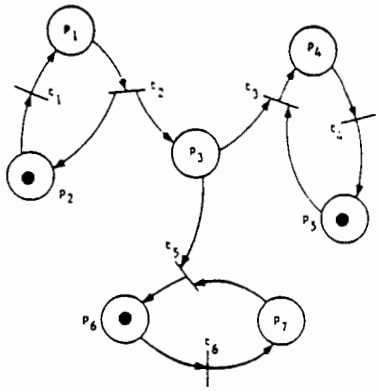

Figure 4: A Sample Petri Net (Figure 4c of [9])

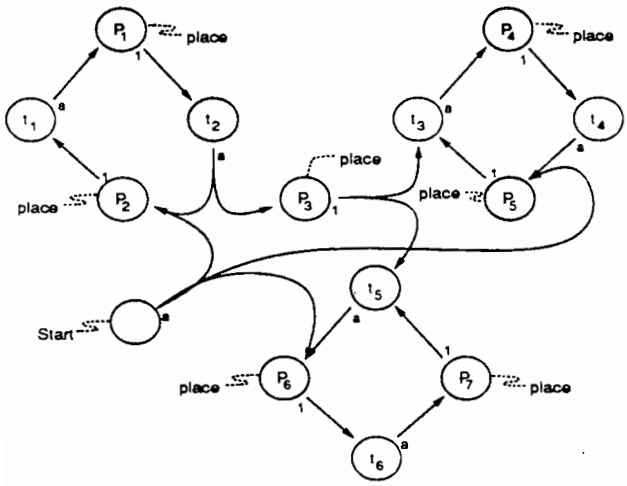

Figure 5: SPG for the Petri Net of Figure 4

(annotations are indicated by dotted curved lines) if they correspond to Petri Net places, and the tails of edges are denoted either "a" or "1", depending on whether tokens flow along that edge to all sinks (parallel edges) or only one sink (nondeterministic edges), respectively.

In going from an SPG to a statechart, we employ the following general heuristics:

- Edges labeled "a" indicate entry into orthogonal components of a statechart. We estimate their nesting by performing a breadth-first search starting at the node annotated "start".
- SPG nodes belonging to the same strongly connected component are assigned to the same orthogonal statechart state; nodes belonging to different strongly connected components are assigned to different orthogonal states.
- Posit the existence of an external event T that constantly occurs.

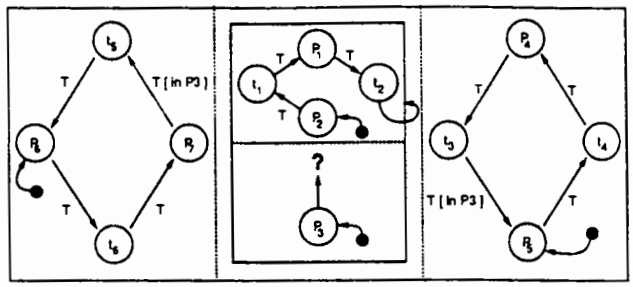

Figure 6: Statechart generated from the SPG in Figure 5

It is important to note that *none* of these heuristics depends on anything to do with a Petri Net or on any information in the SPG that is specific to Petri Nets. Rather, these are generic heuristics, useful for transforming *any* SPG into a statechart. In conjunction with other mappings that are again based on local structural features of the SPG, the result of mapping the SPG into a statechart is shown in Figure 6.

In the statechart, both places and transitions have been translated into statechart states. Three orthogonal states have been created, one of which is further broken down into two more orthogonal states. Each of the three orthogonal states contains one of the "activities" apparent in the original Petri Net. The orthogonal state that is further divided has two more "activities," the P_1-P_2 cycle, and the activity at P_3. When a token is at P_3 something unknown happens – hence the question mark; this is an example of a "black box" in the statechart view.

In fact, this statechart is a fairly good characterization of what is going on in the system. There are three primary activities, one of which does some additional work; the P_1-P_2 cycle not only propagates itself, it also controls P_3, which chooses nondeterministically whether to activate the P_4-P_5 cycle or the P_6-P_7 cycle. P_3 involves a nondeterministic choice, but there is no way to express that directly in statecharts, thus the black box.

Of course, this is probably not the statechart that a developer would create for this system if s/he sat down to create one from scratch, but it certainly presents the salient features of the system in a format that is undeniably a statechart. What is important is that an SPG-based environment can produce such useful information for a system developed using any of a large number of views, not just Petri Nets.

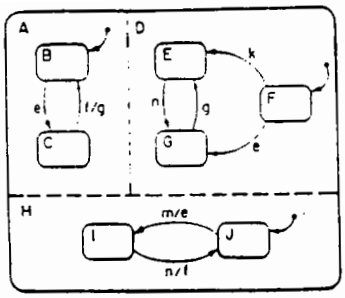

Figure 7: A Sample Statechart (Figure 18 of [10])

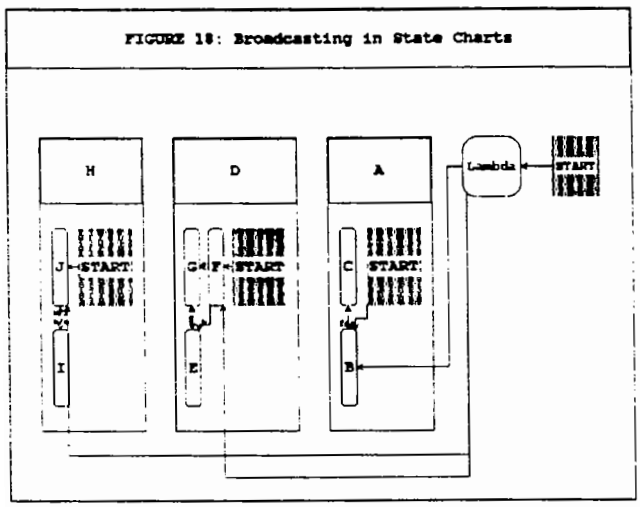

Figure 8: The Statechart of Figure 7 as Produced by Our Prototype

Status

We are in the process of implementing an experimental SPG-based software development environment. In its current state, this environment supports the creation of SPGs and their execution, as well as two views. The first view implements a subset of statecharts, including the discovery of objects in an SPG that "look like" states and events, as well as dynamic visualization of SPG execution. Users can pick statechart objects interactively and obtain information on the underlying SPG objects. Figure 7 shows a sample statechart, and figure 8 shows this same statechart as produced by our prototype.

The second view is a Gelo view of the underlying SPG. Gelo [11] is a graphical layout package, and the primary use of this view is to directly visualize the structure of an SPG. However, Gelo is unable to display the full complexity of an SPG, and the Gelo view has proven to be a useful testing ground for ideas about "black boxing" certain portions of an SPG. Like the statechart view, the Gelo view provides dynamic visualization of SPG execution as well as allowing users to query the system for information about the underlying SPG.

Related Work

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 [12], Gandalf [8], Pecan [13], R^n [14], MENTOR [15], and CENTAUR [16]. 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 [17]. 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 addition, their one-dimensional nature makes them particularly ill-suited for two- or three-dimensional graphical views, which are especially important during specification and design. Similar difficulties arise with program representations based on abstract syntax graphs, such as those used in IPSEN [18] and TEAM [19].

The AMADEUS project [20] has developed a "unified model" of semantics for specifications of software systems, but the model appears to be designed primarily to facilitate batch-mode translations between tool-specific data representations; no interactive tool uses the model directly. The scope of the AMADEUS project seems to be limited to representing specification information – no consideration is given to representing executable programs. Similarly, Lubars, as part of his work on a General Design Representation [21], has proposed a view-independent representation for design information. His research has focused on static design data, however, and has not addressed issues concerning the executability of the resulting representation. Because programs are fundamentally executable entities, we believe it is crucial that a program's representation be executable.

Acknowledgments

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.

References

[1] C. Niskier, T. Maibaum, and D. Schwabe, "A Look Through PRISMA: Towards Plurasistic Knowledge-based Environments for Software Specification Acquisition," in *Proceedings of of the 5th International Workshop on Software Specification and Design*, pp. 128–136, ACM, 1989.

[2] S. P. Reiss, "Working in the Garden Environment for Conceptual Programming," *IEEE Software*, pp. 16–27, Nov. 1987.

[3] S. Meyers and S. P. Reiss, "Representing Programs in Multiparadigm Software Development Environments," in *Proceedings of COMPSAC-89*, pp. 420 – 427, Sept. 1989.

[4] S. Meyers, "Semantic Program Graphs as a Representation for Software Systems." PhD thesis proposal. Available from the authors, June 1990.

[5] S. Meyers and S. P. Reiss, "Semantic program graphs." In Preparation, Sept. 1991.

[6] P. C. Treleaven, R. P. Hopkins, and P. W. Rautenbach, "Combining Data Flow and Control Flow Programming," *The Computer Journal*, vol. 25, no. 2, pp. 207–217, 1982.

[7] Arvind and R. S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Transactions on Software Engineering*, vol. 39, pp. 300 – 318, Mar. 1990.

[8] N. A. Habermann and D. Notkin, "Gandalf: Software Development Environments," *IEEE Transactions on Software Engineering*, Dec. 1986.

[9] J. L. Peterson, "Petri Nets," *ACM Computing Surveys*, vol. 9, pp. 223–252, Sept. 1977.

[10] D. Harel, "On Visual Formalisms," *Communications ACM*, vol. 31, pp. 514–530, May 1988. See also followup letters in the December 1988, May 1989, and August 1989 issues of *Comunications of the ACM*.

[11] S. P. Reiss, S. Meyers, and C. Duby, "Using GELO to Visualize Software Systems," in *Proceedings of UIST '89*, Oct. 1989.

[12] T. Reps and T. Teitelbaum, "The Synthesizer Generator," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, May 1984.

[13] S. P. Reiss, "PECAN: Program Development Systems that Support Multiple Views," *IEEE Transactions on Software Engineering*, pp. 276–285, Mar. 1985.

[14] R. T. Hood and K. Kennedy, "A Programming Environment for Fortran," Tech. Rep. TR84-1, Rice University, June 1984.

[15] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, "Programming Environments Based on Structured Editors: The MENTOR Experience," in *Interactive Programming Environments* (D. R. Barstow, H. E. Shrobe, and E. Sandewall, eds.), ch. 7, pp. 128–140, McGraw-Hill, 1984.

[16] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual, "CENTAUR: the System," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (P. Henderson, ed.), pp. 14–24, ACM Press, Nov. 1988. Published as *ACM Software Engineering Notes* 13:5, November 1988, and *ACM SIGPLAN Notices* 24:2, February 1989.

[17] S. P. Reiss, "GARDEN Tools: Support for Graphical Programming," in *Lecture Notes in Computer Science 244* (R. Conradi, T. M. Didriksen, and D. H. Wanvik, eds.), pp. 59–72, Springer-Verlag, June 1986. Also available as Brown University Computer Science Department Technical Report No. CS-86-18, April 1986.

[18] M. Nagl, "A Software Development Environment Based on Graph Technology," Tech. Rep. 87-3, Lehrstuhl für Informatik III, Rheinisch-Westfälische Technische Hochschule Aachen, Ahornstraße 55, 5100 Aachen, West Germany, 1987.

[19] L. A. Clarke, D. J. Richardson, and S. J. Zeil, "TEAM: A Support Environment for Testing, Evaluation, and Analysis," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (P. Henderson, ed.), pp. 153–162, ACM Press, Nov. 1988. Published as *ACM Software Engineering Notes* 13:5, November 1988, and *ACM SIGPLAN Notices* 24:2, February 1989.

[20] W. J. Black, A. G. Sutcliffe, P. Loucopoulos, and P. J. Layzell, "Translation Between Pragmatic Software Development Methods," in *Lecture Notes in Computer Science 289* (H. K. Nichols and D. Simpson, eds.), pp. 357–365, Springer-Verlag, Sept. 1987. Proceedings of the 1st European Software Engineering Conference, held in Strasbourg, France.

[21] M. D. Lubars, "A General Design Representation," Tech. Rep. STP-066-89, Microelectronics and Computer Technology Corporation, Feb. 1989.