Representing Software Systems in Multiple-View Development Enivironments

Scott Douglas Meyers

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-18
May 1993

Representing Software Systems in Multiple-View Development Environments

Scott Douglas Meyers

B.S., Stanford University, 1983
M.S., Stanford University, 1983
Sc.M., Brown University, 1987

Thesis

Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of
Computer Science at Brown University.

May 1993

For Nancy.
Always for Nancy.
Everything for Nancy.
Nothing without Nancy.

And in memory of my grandfather,
John Percy Briggs, 1904–1992,
who never saw it,
wouldn't have understood it if he had,
but would have been proud just the same.

Abstract

How can software developers be provided with multiple interacting views of the systems they create, debug, enhance, and maintain? That is the fundamental question I address in this thesis.

Related questions immediately suggest themselves. What is a "view" of a software system? Why might developers want to have more than one at a time? How can developers effectively interact with views? How can changes made in one view be propagated to other views? I address each of these questions in this dissertation, but my primary concern is more fundamental: how should the information about a software system be *represented* if multiple views of that system are to be offered?

My answer to this critical question of representation is a new graph structure called Semantic Program Graphs (SPGs). Compared to the representations used in current development environments, SPGs are less biased towards textual depictions of programs, are more expressive of the full range of semantics employed in software systems, and offer more convenient support for the development of custom user-defined views of such systems.

SPGs are the primary contribution of the research described in this thesis. However, there are important additional contributions as well. First, I analyze a number of different approaches to the problem of view integration in development environments, and I offer a comprehensive survey of existing work that places prior research efforts within the framework that grows out of my analysis. Second, I provide a new model for views of software systems that reflects much more realistically the true complexity of systems in practice, and I introduce SPGs as a representation that both supports this model and can act as an effective integration mechanism for multiple views. Finally, I provide an architecture for an interactive multiple-view development environment built around SPGs.

Contents

Acknowledgements **xi**

I Multiple-View Software Development **1**

1 Introduction **1**
1.1 Views and Presentations 1
1.2 Multiple Views 3
1.3 Integrating Views 4
1.4 Scenario for an Ideal Environment 6
1.5 Contributions of this Research 9
1.6 Organization of this Thesis 11

2 Approaches to View Integration **13**
2.1 Integrating Multiple Views 13
2.2 Five Approaches to Integration 15
2.2.1 Shared File System 15
2.2.2 Message Passing 16
2.2.3 Simple Database 19
2.2.4 View-Oriented Database 21
2.2.5 Canonical Representation 22
2.3 Discussion 24
2.4 A Categorization of Existing Systems 25
2.4.1 Multiparadigm Development Environments 26
2.4.2 MVDEs Not Using a Canonical Representation 27
2.4.3 MVDEs Using a Canonical Representation 29
2.5 Conclusions 31

3 Designing a Representation **33**
3.1 The Limits of Mapping Between Views 34
3.2 Paradigms For Software Development 35
3.3 A Model for Views of Software Systems 36
3.3.1 The Sequential Control Flow Paradigm 37
3.3.2 The Dataflow Paradigm 39
3.3.3 The Parallel Control Flow Paradigm 40
3.3.4 Summary 41

3.4 The View Model and Data Structures 41
3.5 Implementing the View Model 42
3.5.1 Choosing Semantic Primitives 43
3.5.2 Representing Iterative Loops 44
3.5.3 Representing Subprograms and Calls 45
3.5.4 Representing Scopes and Name Bindings 45
3.5.5 Representing Interprocess Communication 48
3.6 Summary of Constraints 49
3.7 An Evaluation of Existing Canonical Representations 50

II Semantic Program Graphs 53

4 Semantic Program Graphs 53
4.1 Overview 54
4.2 Unexecutable Components 60
4.3 Executable Components 62
4.3.1 Nodes 62
4.3.2 Edges 78
4.3.3 Groupings 85
4.4 Annotations 95
4.5 Execution 100
4.6 A Simple Example 107
4.7 Summary 109

5 SPGs and the View Model 111
5.1 Representing Model Features Using SPGs 111
5.1.1 The Sequential Control Flow Paradigm 111
5.1.2 The Dataflow Paradigm 112
5.1.3 The Parallel Control Flow Paradigm 113
5.2 Solving Representative View Problems 113
5.2.1 Representing Iterative Loops 113
5.2.2 Representing Subprograms and Calls 118
5.2.3 Representing Scopes and Name Bindings 119
5.2.4 Representing Interprocess Communication 119

6 Using SPGs 124
6.1 An Architecture for an SPG-Based MVDE 124
6.2 Communication Between PMs and the SPGM 126
6.2.1 The SPGM Interface 127
6.2.2 The PM Interface 130
6.3 Updating Views 134
6.4 Mismatches Between SPGs and Views 135
6.4.1 Missing Features in a View 136
6.4.2 Differences in Semantics for Common Features 138
6.4.3 Incompatibility Between a View and the View Model 139
6.5 Error Detection 139

6.6 Suitable Views . . . 141
6.6.1 Source Code . . . 142
6.6.2 Flowchart . . . 143
6.6.3 Call Graph . . . 143
6.6.4 Runtime Stack . . . 143
6.6.5 Resource Profile . . . 144
6.6.6 Test Coverage . . . 144
6.6.7 Build Dependencies . . . 145
6.6.8 Petri Net . . . 145
6.6.9 Conclusion . . . 145
6.7 Experience with a Prototype Implementation . . . 146
6.7.1 Ancillary Research on Object-Oriented Programming . . . 151
6.8 Summary . . . 153

7 A Preliminary Assessment 154
7.1 A Paper Experiment . . . 154
7.2 Translating an FSA into an SPG . . . 157
7.3 Translating Pascal into an SPG . . . 160
7.4 Translating an SPG into an FSA . . . 164
7.5 Translating an SPG into Pascal . . . 168
7.6 Translating an SPG into a Runtime Stack . . . 174
7.7 Development of a Multiparadigm Program . . . 176
7.8 Conclusions . . . 177

8 Topics for Further Research 183
8.1 Further Work on SPGs . . . 183
8.2 Related Research Areas . . . 186

III Appendices and Bibliography 189

A Mapping Between Petri Nets and Ada Tasks 189
A.1 Mapping Between Petri Nets and SPGs . . . 189
A.2 Mapping Between Ada Tasks and SPGs . . . 194
A.3 Viewing Ada Tasks as Petri Nets . . . 198
A.4 Viewing Petri Nets as Ada Tasks . . . 204

B Pseudocode for Sample Mappings 211
B.1 Translating an FSA into an SPG . . . 212
B.2 Translating Pascal into an SPG . . . 216
B.3 Translating an SPG into an FSA . . . 225
B.4 Translating an SPG into Pascal . . . 237
B.5 Translating an SPG into a Runtime Stack . . . 259
B.6 Modifying the Action Routine for an FSA State . . . 262

C SPG-Related Publications 265

Bibliography **268**

Acknowledgements

There is a defining moment in each person's life — or at least there should be — when that person realizes that, contrary to all expectation, he or she does *not* know everything. For me, this moment came courtesy of Gene Cordell, and I am indebted to him for it. Had he not challenged my youthful assumptions, my journey through graduate school would never have begun.

My advisor, Steve Reiss, introduced me to the idea of multiple-view software development, and many of the themes that run through my research can be traced back to his comments and suggestions. His influence is especially evident in the organization of this dissertation, which is substantially stronger now than it was when I first showed it to him. The other members of my committee — Peter Wegner, Stan Zdonik, and Pamela Zave — also improved this final version of the manuscript through their careful reading, insightful observations, and uncompromising demands.

John Stasko and Gail Mitchell contributed comments on early drafts of individual chapters of this thesis. George Nelan contributed the Lucid program that appears on page 39, Fred Wild and Carolyn Duby provided me with the `with` clauses necessary to breathe life into the Ada program of Appendix A, and Tom Christiansen provided me with information about assignments in Perl. Ted Goldstein's comments persuaded me to remove a report on some additional research that I had planned to include as yet *another* appendix.

Years ago I had the hubris to claim I would be one of the few students of Computer Science to spend more time performing their doctoral research than typesetting the document describing it. Whether I achieved that goal is unclear, but this much is certain: were it not for the assistance of Richard Hughey, Paul Howard, Gerd Hillebrand, Brook Conner, and Darren Erik Vengroff, I'd still be struggling to get LaTeX to produce even an approximation of what I wanted. Of course, the task would have been hopeless from the outset had Alex Shvartsman not bequeathed his LaTeX manual to me.

I have always had the good fortune of being surrounded by an excellent computing environment at Brown University. Even more important, I have been surrounded by an excellent technical staff, the rarity of which has been attested time and again by distraught alumni as they struggle to adjust to their new positions, both in academia and in industry. The members of the Computer Science Department's tstaff have gone out of their way to help me more times than I care to remember, and I am pleased to be able to acknowledge their efforts on my behalf. Similarly, it has been a singular pleasure to work with the members of the department's administrative staff (astaff).

One of many contradictory aspects of being a doctoral student is that although the performance of the research — and certainly the physical act of writing the dissertation that grows out of it — is an intensely solitary experience, life as a graduate student itself

can be rather social. There is especially a sense of camaraderie that develops amongst members of a graduate student office, and over the years, I have shared offices (hence commiserated) with a number of other students. Their companionship has helped smooth the way along a path featuring more than its fair share of bumps. I am especially thankful to Gail Mitchell, Richard Hughey, Andrea Skarra, and Alex Shvartsman for making the trek more tolerable. Non-officemates who have enlivened the journey include Kathy Kirman, Tim Johnson, Karen Bowen, Anne Leinster, Tim Farley, Simon Kao, Jill Huchital, and virtually the entire staff of Store 24.

I have read a fair number of dissertations in my day. More, in fact, than would probably be considered prudent by members of the medical profession. Possibly because it reveals more of the person behind the document than any other part, I have always found myself drawn to the acknowledgements, and rare indeed is the author who fails to acknowledge the support of his or her spouse with some degree of poignancy. I initially viewed this convention as mere obligation, but I now know better. It is a simple fact that this dissertation *would not exist* without the constant encouragement, steadfast determination, endless patience, and selfless devotion of my wife, Nancy L. Urbano. If being in a Ph.D. program is a trying experience (and it is), it is doubly trying to suffer through it vicariously as a spouse. It is impossible to accurately characterize my debt to Nancy; suffice it to say I owe her for life.

My parents, Phyllis and Douglas Meyers, have waited — sometimes patiently, sometimes anxiously — for my tenure here to expire, and I am grateful to them for their support these past eight years.

Financial support for the research described in this dissertation was provided by the NSF under grants DCR 8605567, CCR 9111507, and CCR 9113226; by ONR under contract N00014-83-K-0146 and ARPA order 6320; by ONR grant N00014-91-J-4052 and ARPA order 8225; and by the Digital Equipment Corporation under agreement 393. Dave Sklar and Donald French helped improve a dreary financial landscape by giving me my introduction to consulting, and the Trust Group of the United States National Bank of Oregon provided assistance by agreeing to defer repayment of my student loans for *much* longer than they ever anticipated.

Part I

Multiple-View Software Development

Chapter 1

Introduction

How can software developers be provided with multiple interacting views of the systems they create, debug, enhance, and maintain? That is the fundamental question I address in this thesis.

Related questions immediately suggest themselves. What is a "view" of a software system? Why might developers want to have more than one at a time? How can developers effectively interact with views? How can changes made in one view be propagated to other views? I address each of these questions in this dissertation, but my primary concern is more fundamental: how should the information about a software system be *represented* if multiple views of that system are to be offered?

My answer to this critical question of representation is a new graph structure called Semantic Program Graphs (SPGs). Compared to the representations used in current development environments, SPGs are less biased towards textual depictions of programs, are more expressive of the full range of semantics employed in software systems, and offer more convenient support for the development of custom user-defined views of such systems.

SPGs are the primary contribution of the research described in this thesis. However, there are important additional contributions as well. First, I analyze a number of different approaches to the problem of view integration in development environments, and I offer a comprehensive survey of existing work that places prior research efforts within the framework that grows out of my analysis. Second, I provide a new model for views of software systems that reflects much more realistically the true complexity of systems in practice, and I introduce SPGs as a representation that both supports this model and can act as an effective integration mechanism for multiple views. Finally, I provide an architecture for an interactive multiple-view development environment built around SPGs, and I describe a preliminary prototype implementation of the architecture.

1.1 Views and Presentations

A *software system* is a generalized notion of a program, one that encompasses more than just executable information. For example, a software system might consist of both its executable information (the "program") and its attendant comments and documentation. A software system might also consist of more than one program, such as in a system consisting of multiple independent processes. However, a software system need not be a complete

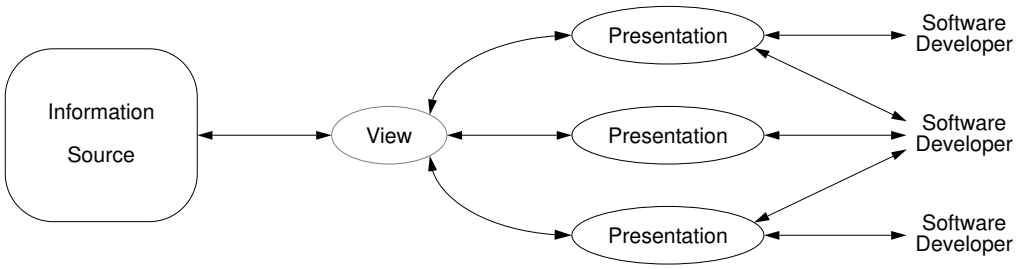


Figure 1-1: Relationship among an information source for a software system, a view of that system, presentations of the view, and software developers.

program. A subroutine library, for example, is also a software system. In this thesis, I usually focus on executable information and on traditional program models that start with a single flow of control, so I often use the terms "program" and "software system" interchangeably, but I tend to rely on the latter, more general, term to emphasize the fact that I am concerned with much more than just executable information about single processes.

Given a source of information about a software system, that is, a database or data structure of some kind, a *view* of the software system is a set of mappings between the information source and some *virtual* information source that is more appropriately structured for a particular use. For example, a call graph is one possible view of a program, one in which the only aspects of the software system that are important are routines and calls to routines. In the case of a call graph, the set of mappings corresponding to the view is likely to be equivalent to a database projection, but in general, views may be much more complex than simple projections.

Views may be static or dynamic, and so may the information sources from which they are derived. Static views of a software system are unaffected by execution of that system, while dynamic views change in some way as the system runs. The dynamic aspects of a dynamic view may be simple and straightforward, such as highlighting the routine currently being executed in a graphical call graph, or may be arbitrarily complex, such as a hand-crafted algorithmic animation [30, 194, 191]. Some views, such as source code, may be editable. Other views, such as the output from a program profiler, may be inherently read-only. This definition of a view is consistent with that of updatable views in the database world [105].

Software developers interact with a view through one or more *presentations* of that view. A presentation is a pair of mappings, one from a view to operations on some set of output devices, and one from operations on some set of input devices to operations on a view. Presentations may manifest themselves textually (e.g., traditional source code), graphically (e.g., a traditional call graph), through unconventional media (e.g., aurally [96, 63, 125]), or, frequently, some combination of these. The relationships among an information source for a software system, a view of that system, presentations of that view, and software developers is summarized in Figure 1-1. As shown in the figure, it is entirely possible for a single developer to simultaneously use more than one presentation or for a single presentation to be simultaneously employed by more than one developer.

It is often useful to think of a view as an abstract syntax for a language describing the underlying software system, and of a presentation as a concrete syntax for the language. In

fact, this is the basic approach of the Visual Programmer's Workbench [72], which supports the creation of language-specific development environments for visual languages.

In practice, researchers tend to ignore the differences between views and presentations, generally referring only to "views" unless context forces them to draw an explicit distinction. While there is almost universal agreement among researchers in the field of relational databases that a view is equivalent to a query [213],[1] there is no corresponding agreement for the term in the field of programming environments.[2] As a result, there is an advantage to speaking informally: most everyone agrees on *roughly* what a view is, even though not everyone may agree on *precisely* what it is. In this thesis, then, I tend to follow the convention of blurring the distinction between views and presentations, as the looser usage is reflective of the way people usually speak and write about these terms.

1.2 Multiple Views

Software systems are complicated. Because they are complicated, they are difficult to explain to others, and they are difficult to comprehend. Developers invariably use visual means to convey information about how a system works, but no single visual depiction can hope to capture the intricacy of a large system. As Brooks has noted,

> The reality of software is not inherently embedded in space. ... As soon as we attempt to diagram software structure, we find it to constitute not one, but several, general directed graphs superimposed one upon another. The several graphs may represent the flow of control, the flow of data, patterns of dependency, time sequence, name-space relationships. These graphs are usually not even planar, much less hierarchical [28, p. 12].

In recognition of this inherently multi-faceted nature of software, many researchers in the area of software development environments have argued that environments that support only a single view — usually textual source code — are inadequate [121, 113, 124]. In fact, software developers already routinely employ multiple views of the systems they work on. One need only note the plethora of more or less "standard" views to understand that multiple-view software development is already the rule in practice. Common views include source code, flowcharts, call graphs, program profiles, class hierarchies, Petri nets , statecharts, dataflow diagrams, state transition networks, object graphs [23], module interconnection diagrams, algorithm animations, test coverage analyses, dependency graphs, debuggers, program slices, and many others.

Environment support for program development has traditionally focused on textual depictions of programs — usually source code — but it's important to note the increasing interest in nonlinear programming languages. Examples of this less conventional approach to software development include structured analysis [177], Fabrik [97], statecharts, grids [149, 150], and graphical interfaces to conventional languages [55]. Further examples can be found in the surveys by Raeder [160], Myers [138], and Davis [47]; the introductory books

[1] The question is still open in the field of object-oriented databases, however [184, 86].

[2] For the purposes of this thesis, the terms "programming environments," "software development environments," "software engineering environments," etc., are all synonyms. I am concerned with issues that affect both single-developer endeavors (programming-in-the-small) as well as projects arising from teams of developers (programming-in-the-large).

by Shu [185] and Chang [37]; and the proceedings of the IEEE workshops on visual languages (held in 1984 and annually since 1986). The result of this interest is that the number and types of views available to software developers is continually increasing.

Certainly, then, multiple views are natural. Perhaps less obvious is the notion that the set of views available to developers should be open-ended, that developers should themselves be able to define new types of views. The basic argument is straightforward: it is unrealistic to expect the designer of a development environment to correctly anticipate all views that developers might want, so any fixed set of views must prove too limiting for some users of the environment. In fact, Lehman argues that for E-type software [115] (which encompasses development environments), this problem is *inherent*, a consequence of "Heisenberg-like uncertainty" that cannot be overcome [116, 114]. It is therefore *essential* for users to be able to define their own views if the environment is to remain satisfactory.

The researchers who developed Voyeur, a system for generating views of the state of parallel computations, make a less formal, but similar argument:[3]

> The Voyeur prototype simplifies the task of building views. If these costs are too great, programmers will be discouraged from constructing views appropriate for a given program. ... Since each parallel program might require its own view or views, it must be feasible for all parallel programmers, not just specially trained experts, to construct views effectively [191, p. 1].

In sum, multiple-view software development emerges naturally from the intrinsically multi-faceted nature of software systems, is already embraced by programmers as a matter of current practice, and can be theoretically defended. Yet few current development environments offer explicit support for multiple views, and almost none have as a primary concern the ability of developers to easily define custom views. (The sole exceptions are Voyeur and Garden [164].) The research described in this thesis directly addresses these shortcomings of current programming environments.

1.3 Integrating Views

It is useful to have multiple views available, but it is equally useful to work with more than one view at a time. For example, a developer might want to work with a high-level abstract view (such as a call graph) to get an overall feel for a system's structure, while simultaneously using a more detailed view (perhaps source code) for particular parts of that system. Similarly, given a large system being worked on by a team of developers, each developer may wish to employ a unique view that best suits the activities in which she or he is engaged.

This leads to the problem of view integration. If, through one view, a programmer makes a change to the system being worked on (e.g., adds a statement, deletes a function, reformats a comment), how — if at all — should that change be reflected in other views? In other words, how should different views of the same system be kept consistent?

[3] One might be inclined to dismiss the Voyeur experience as inapplicable to conventional, serial, programs, but the authors note that "Voyeur, while motivated by the complexity of debugging parallel ... programs, is attractive for debugging complex sequential programs as well."

This integration problem is a prime concern for designers of multiple-view development environments (MVDEs). I examine it in detail in Chapter 2, but it's worth outlining the basic options here:

- **Don't worry about integration.** Different views are offered by different tools, and these tools run independently, unaware of the presence of other tools that may be offering alternative views of the same system. This approach, characteristic of loosely coupled toolbox-based environments like Unix, is by far the most common one in practice.

- **Make all views orthogonal.** Different views are carefully designed to show only orthogonal information, so the integration problem simply vanishes: it is impossible for a change made in one view to manifest itself in any other view. In practice, software systems are so laden with interdependent features that this approach is unworkable; truly orthogonal views suffer from so many restrictions that they aren't worth the trouble.

- **Use only a single view, but multiple presentations.** The more similar the views, the less daunting the integration problem. Given that a view can be considered analogous to abstract syntax and a presentation analogous to concrete syntax, one approach to integration is to allow only one view, but multiple presentations of that view. For example, a call graph might be depicted both graphically and in a tabular who-calls-who form. This reduces the view integration problem to that of mapping between abstract and concrete syntaxes (i.e., parsing and unparsing), but limits the range of views that can be integrated.

- **Find some way for views to communicate with one another.** When a developer changes a system through a view, other views are somehow told about the change, and these views do whatever is necessary to make themselves consistent with the modified system. This simple idea turns out to be difficult to implement, but a consensus is forming in the research community that this is the "right" way to achieve view integration.

The problem of view integration in the presence of editable views is hardly confined to programming environments. Database researchers have grappled with it for years. Fundamentally, the difficulty boils down to underspecification. Keller offers this example:

> Consider an *Employee* relation with Employee and Department attributes, and a *Department* view with Department and Number of Employees attributes. The view update request to increment the number of employees in the Computer department would result in the need to add an unknown employee to the *Employee* relation. A request to decrement the number of employees in the Toy department would require the computer to choose an employee to fire – a decision best done by some person [105, p. 64–5].

Although some researchers have investigated semantics-based view updates [105], the most common way to deal with this problem is to keep it from happening in the first place. This is typically achieved by restricting the kinds of modifications that can be made through database views. As a general mechanism for MVDEs, however, this strategy is too

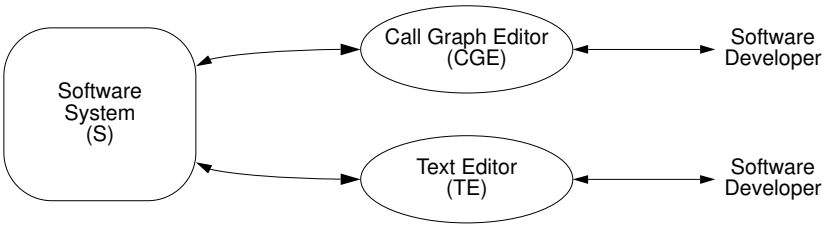


Figure 1-2: Development environment for a software system with views as a text editor and a call graph editor.

inflexible, because limiting view-based changes to a system to those that are unambiguous may well lead to editable views with no more utility than read-only views. For example, if an edge is added to a call graph view of a system, the location of the newly added call in the code of the calling node is underspecified, but it is still an editing operation that makes sense. Support for such permissive editing was an important factor in my design of SPGs and the MVDEs that use them.

There are other important aspects of view integration that must also be taken into consideration when coming up with an architecture for a development environment; the appropriate granularity for propagating changes, for instance. I investigate these additional considerations in Chapter 6.

1.4 Scenario for an Ideal Environment

What would it be like to create and maintain software in an ideal MVDE? Consider a developer working in an environment with two integrated views. One view takes the form of a conventional text-based program editor. This view is called TE ("Text Editor"). The other view offers a depiction of the system as a call graph. Unlike most call graph displayers, however, this one is a full-fledged editor, meaning that routines and calls can be added and/or removed from the system through its interface. It is called CGE ("Call Graph Editor"). Both TE and CGE offer views of a single underlying software system, S (see Figure 1-2). As I noted earlier, the difference between views and presentations tends to be ignored in informal parlance, and Figure 1-2 follows this convention: the TE view and presentation and the CGE view and presentation have been merged into single elements.

TE and CGE both support dynamic views of S by highlighting portions of what they display. TE highlights the line of source code currently executing, while CGE highlights the routine in which that code is located. CGE also shows which routines are currently on the stack by highlighting them in a different color, behavior that can be found in the call graph display tool of the FIELD programming environment [165]. If S exhibits parallelism, then both TE and CGE might highlight more than one thing at a time during execution of S.

In this idealized environment, software development might proceed as follows:

1. A software developer examines S through TE, and the developer notices that two of the routines in S, R_1 and R_2, have common code. The developer decides to turn this common code into a more general routine, R_3, then have R_1 and R_2 each call R_3.

2. The developer uses CGE to add R_3 to S. The new routine immediately shows up in TE.

   This conceptually simple operation entails dealing with a large number of details. For example, if TE is displaying a strongly typed language like Ada, how — if at all — are the parameter types and return type of R_3 determined for display in TE? Similarly, where in the textual view of the program displayed in TE should R_3 be placed?

   There are a number of approaches to these questions. CGE might prompt the developer for the signature of R_3 when it is added; CGE might use default values (e.g., an empty parameter list, a null return type); or it might leave the information unspecified, leading to an incomplete entry for R_3 in TE.[4] Similarly, the location of R_3 in TE might be user-specified through CGE, or it might default to the top or bottom of the file that TE is displaying.

   TE is likely to be passive during this operation, because only the developer using CGE understands the purpose of adding R_3 to S. If the same developer is also using TE, then that developer might choose to add the routine to S via CGE and then specify the full signature via TE, but it is unreasonable for TE itself to provide any default information about R_3, because the developer using TE might well be different from the developer adding R_3. It would be unreasonable, for example, for TE to prompt its user for information about the newly appearing routine R_3, because that developer might have no idea why some other developer was adding R_3 in the first place.

3. Still using CGE, the developer modifies R_1 and R_2 so that they both call R_3. The new call sites are automatically added to the file being edited by TE.

   Again, important questions surround this seemingly straightforward modification. In the TE view of R_1 and R_2, where do the calls get added? What values are used as actual parameters in the calls? How many calls from R_1 and R_2 to R_3 are added? As before, the answers to these questions are likely to be provided by CGE or the developer using it, because there is no way for TE itself to supply the answers, and users of TE may not be in a position to provide the necessary information.

4. Using TE, the developer copies code from R_1 and R_2 to R_3, making whatever adjustments are necessary. Suppose it turns out that part of the code common to R_1 and R_2 is a call to a routine R_4. Then the call to R_4 is moved from R_1 and R_2 to R_3. During this operation, the CGE view of S is automatically updated: edges from R_1 and R_2 to R_3 disappear, and a new edge appears between R_3 and R_4.

   In this case, communication between TE and CGE is as straightforward as it sounds. Because the CGE view of S depicts a proper subset of the information present in a TE view of S, there is no ambiguity about the changes to be made to CGE in response to changes made to S via TE.

[4] Most current environments allow programs to be syntactically invalid while they are being edited, because experience with structure editors demonstrated that forcing users to maintain syntactically valid programs during editing was an unacceptable restriction [218]. The COPE environment [44], however, was a novel experiment in that the responsibility for ensuring that a program was always grammatically valid fell on the *environment*, not the user.

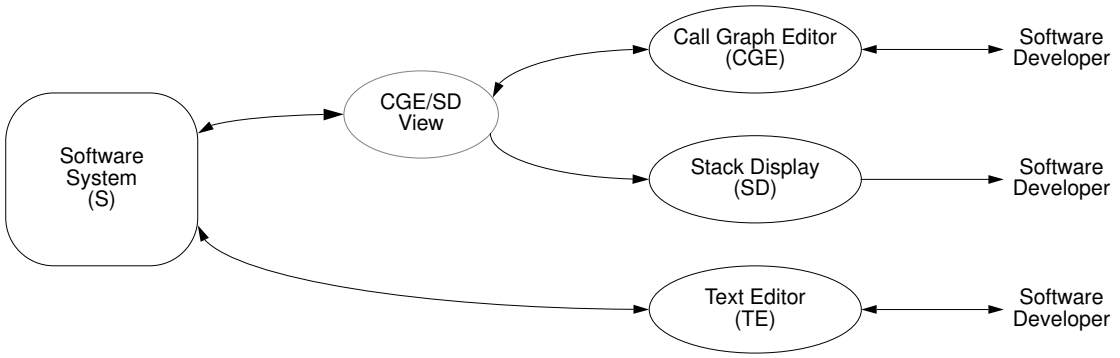


Figure 1-3: Development environment augmented with stack display presentation and with the call graph and stack view explicitly shown.

5. Having made the changes just described, the developer reexecutes the system. However, the developer is unhappy with the implicit view of the runtime stack(s) shown in CGE. Although it is easy to see which routines are on a stack, it is difficult to know their relative ordering, and it is impossible to tell if a routine is on a stack more than once or is on more than one stack. The developer decides to write a new dynamic view of S to display the runtime stack(s) in a more comprehensible format.

   This new view — call it SD ("stack displayer") — is likely to be implemented as a new presentation of the information already in the view used by CGE. This is indicated in Figure 1-3, in which the view shared by the presentations CGE and SD is made explicit. In the figure, the unidirectionality of the arrows around SD indicate that it is a read-only presentation; it is impossible to modify S through SD.

6. The developer creates SD, which displays a textual stack of routine names (or, for parallel applications, a set of stacks), the topmost name on each stack being the name of the routine currently being executed. When the developer reexecutes S, the new stack view is automatically integrated into the environment. This is due to the fact that SD is really just a different way of presenting the information in the CGE view. Because the CGE view already "knows" when routines are added to or removed from a stack, that information is readily available to SD.

   How does the developer come up with the code for SD? Perhaps the developer writes it from scratch. More likely, as suggested above, the developer bases it on the code for a preexisting view or presentation. If the environment is implemented using object-oriented technology, the developer may derive new classes from existing classes and then specialize the behavior of inherited functions (see Section 6.7). If the environment is implemented using more conventional technology, the developer may simply copy some existing code and then make the necessary modifications.

   For the purposes of this thesis, the mechanism employed for the creation of SD is irrelevant. What *is* relevant is that new views can be easily created only if the appropriate information about the system — in this case, S — is readily available and if there is an integration mechanism in place that coordinates the actions of different views. My research, therefore, concentrates on addressing these more fundamental

issues, rather than on the physical act of writing views. This is in contrast to the Voyeur system [191], for example, which presupposes the existence of a suitable information source and integration mechanism; it is primarily concerned with facilitating the physical production of views.

This scenario provides a sense of what it might be like to develop software through multiple interacting views. At the same time, it glosses over a host of interesting and important questions. For example, when should a change made in one view be propagated to other views? What will happen if more than one view at a time modifies a software system? I consider these questions again in Chapter 6, where I discuss the pragmatics of an SPG-centered MVDE.

1.5 Contributions of this Research

The technical issues confronting the designer of an MVDE are numerous, difficult, and intertwined. It is worth recapping the primary challenges here:

- Multiple views, multiple presentations, and multiple developers.
- Views and presentations that may be static or dynamic, read-only or read-write.
- Both predefined and user-defined views and presentations.
- Automatic consistency maintenance between views during editing.

I can hardly claim to have fully resolved each of these issues, but the results of my research increase our understanding of them, and it buttresses our ability to design and implement software development environments that approach the behavior of the idealized environment described in Section 1.4. My contributions are summarized below and are more fully described in the sections that follow.

- An **analysis** of the suitability of different approaches to view integration for MVDEs, including a comprehensive categorization of existing systems.
- A **model** for views that directly supports a much wider variety of semantic features than do existing development environments.
- A novel **representation** for software systems that supports the view model and that is designed specifically for use as the core of an MVDE.
- An **architecture** for interactive multiple-view development environments based on my representation, including a simple prototype implementation.

An Analysis of Approaches to View Integration

Within the research community there is general agreement that view integration is a goal worth striving to attain, but there is no consensus regarding how this integration should be accomplished. Over the course of the last decade, dozens of systems have been designed and/or created that offer greater or lesser degrees of integration, and this integration has been achieved through any number of technical devices. Such devices include shared file

systems, explicit bidirectional mappings between views, traditional (typically relational) databases, special databases designed specifically to support multiple views, message passing between separate processes, and more. Each of these integration mechanisms offers a unique set of advantages and disadvantages, both in terms of the issues discussed in this chapter, as well as ancillary issues such as the ability of an environment to gracefully evolve over time.

What has been missing from this experimental extravaganza, and what I provide in this thesis, is an *analysis* of the different approaches to view integration. Though there are scores of implementations, most are variations on a small number of recurring themes, and I identify and describe the five fundamental integration mechanisms employed to date. I also examine each of these integration mechanisms in terms of its suitability for use in an environment dedicated to supporting multiple interacting views, and I explain why a canonical program representation is the integration mechanism of choice. Finally, I categorize 74 systems in terms of their primary integration mechanism. My analysis provides a framework in which to evaluate the strengths and weaknesses of integration mechanisms for MVDEs, as well as a map of the conceptual terrain already explored by the collective efforts of the research community.

A Model For Views

Taken as a group, programming languages support an enormous variety of syntactic and semantic features. However, mainstream languages like FORTRAN, COBOL, Pascal, C — even Lisp — support only a fraction of these features. Their concrete syntax is always textual, and their underlying model of computation is invariably serial, deterministic, control-driven, synchronous, and strict.[5] Such languages are a poor reflection of the true range of executable systems, which may employ a nontextual, nonlinear concrete syntax and which may contain constructs supporting concurrent, nondeterministic, data-driven, asynchronous, and/or nonstrict semantics.

Unfortunately, nearly all the work on programming environments has implicitly adopted the restricted syntactic and semantic model of these mainstream programming languages and has been designed to support views built atop this model. In this thesis, I present a new model for views, one in which there is no presupposition that the concrete syntax is textual and linear, and one in which semantic support for concurrency, nondeterminism, asynchrony, and other "uncommon" features is assumed from the start. This model dictates a set of design constraints for a representation for software systems in an MVDE, because any such representation must be able to support the model for views I present. These constraints in turn ensure that my research is applicable to a wide range of programming languages, including visual languages and parallel, nondeterministic, and data-driven systems.

A Novel Representation for Software Systems

Most work on MVDEs has been based on supporting development in mainstream languages, so the syntactic and semantic features supported by such environments almost always correspond to the restricted model of executable systems those languages imply. As a result,

[5] Informally, a *strict* language is one in which the expressions passed to a called routine (i.e., the actual parameters) are evaluated even if their values are not needed. A formal definition of strictness is presented in Section 3.3.

the representations for software systems used in existing MVDEs are unable to support the view model I introduce in this thesis. This is not surprising: my work on extending the model naturally leads to a need to redesign a representation for systems compliant with that model.

The representation I developed — the Semantic Program Graph — has a number of important features that make it appropriate for representing software systems in MVDEs. First, in comparison to traditional program representations like abstract syntax trees, it is much less biased toward linear concrete syntaxes. This makes it more suitable for representing visual and other untraditional languages. Second, it is semantically richer, providing explicit support for both control- and data-driven execution, both serial and parallel flows of control, and both deterministic and nondeterministic execution. Its model of routine calls allows for multiple return values as well as both strict and nonstrict parameter passing. Third, it identifies and enforces a division between view-dependent semantics and view-independent semantics in an MVDE, a natural division that corresponds to the division between stores and environments in the denotational semantics of programming languages [208]. Fourth, it allows for a more flexible block and scoping structure than is offered in traditional languages. For example, it is possible for a portion of a program to be nested inside two or more different scopes, none of which have any nesting relationship with the others. Finally, it explicitly provides for the representation of unexecutable information, such as program commentary and view-specific information, and it allows this ancillary information to be associated with arbitrary and flexibly-defined subgraphs of an SPG. For example, a comment that indicates a relationship between two or more components of a program can be easily associated with each of the components, rather than being forced to exist in only one place in the program.

An Architecture for Multiple-View Development Environments

My final contribution is a general software architecture for MVDEs based on SPGs. This architecture explicitly incorporates an SPG as an abstract data type, an SPG Manager as the interface of the abstract data type, views as virtual data structures, and presentations for interacting with developers. Automatic view integration is achieved through the use of a shared data structure between views (the SPG), and communication between views is ensured by having a single agent managing changes to the shared data structure (the SPG Manager). The creation of custom views is facilitated by the fixed and well-defined interface to SPGs (the SPG Manager), and new views are automatically integrated with existing views because the SPG Manger notifies all views of changes to an SPG whenever such a change occurs.

In addition to developing this architecture and analyzing its strengths and weaknesses, I also implemented a simple prototype MVDE based on SPGs. This prototype served as a simple testing ground for my ideas about SPGs, and it also led to a number of interesting observations about object-oriented programming and C++ software development.

1.6 Organization of this Thesis

In Chapter 2, I examine a number of approaches to the problem of integrating multiple interacting views in a single development environment. I describe the issues an integration

mechanism must address, and I summarize the advantages and disadvantages of various approaches to integration. I provide a comprehensive survey of work in the area of MVDEs, categorizing 74 systems on the basis of the primary integration mechanism they employ. Finally, I justify my decision to focus on a canonical representation as the most appropriate integration mechanism for interactive multiple-view development environments.

Chapter 3 examines the issues confronting the designer of a canonical representation, and I develop a new model for views of software systems that is derived from three fundamental computational paradigms: sequential control flow, dataflow, and parallel control flow. In this chapter I also introduce four problems that are representative of those that must be handled by a canonical representation, and I describe a general approach to determining the features a canonical representation should offer.

I formally define SPGs in Chapter 4. There I describe each structural component of an SPG, define its semantics, justify its existence, and give simple examples of its use. This chapter also provides the algorithm for executing SPGs.

Chapter 5 ties SPGs back to the view model of Chapter 3. First, I show how SPGs fulfill the expressiveness criterion dictated by the view model. Second, I demonstrate how SPGs can be used to deal with the four problems introduced in Chapter 3 as representative of those that a canonical representation must be able to handle.

The topic of Chapter 6 is how SPGs could be used as the linchpin of a multiple-view development environment. There I provide a detailed description of the SPG Manager and how it is used to bring about change propagation from view to view. I also examine how view writers might cope with problems arising from mismatches between SPG semantics and view semantics. I conclude the chapter with a description of the prototype system I developed, and I summarize how this prototype led to interesting research results in the area of object-oriented programming.

Chapter 7 reports on a paper experiment I undertook to better assess how effective SPGs are likely to be in practice. The experiment consisted of developing complete view-SPG and SPG-view mappings for two different views (Deterministic FSAs and a subset of Pascal), as well as the creation of an additional mapping from SPGs to a dynamic view of the runtime stack of a software system as it executes. The experiment also included a component whereby I employed a multiparadigm development methodology to create a single program using more than one view. My experiences during the course of this experiment contributed to series of preliminary conclusions regarding the strengths and weaknesses of SPGs as a practical method to achieve integrated development environments offering multiple views.

Chapter 8 outlines some directions in which my research on SPGs could be extended. It also discusses other research topics that, though not specifically related to SPGs, address important issues in the area of multiple-view software development

There are three appendices. Appendix A gives an informal treatment of the issues involved in developing an SPG-based mapping between Petri nets and Ada tasking constructs; Appendix B contains the complete source (pseudo)code for the mappings I developed for the experiment I describe in Chapter 7; and Appendix C summarizes the publications that have grown out of my work on SPGs and multiple-view software development.

I conclude this dissertation with an extensive bibliography.

Chapter 2

Approaches to View Integration

In this chapter I explore the demands that are made on the integration mechanism in a MVDE, I identify five fundamental approaches to integration, and I examine how well these five approaches meet the demands. I also provide a comprehensive survey of work in the area of MVDE integration, categorizing 74 systems on the basis of the primary integration mechanism they employ.

2.1 Integrating Multiple Views

What does it mean for an environment to be integrated? An integrated environment might be one in which all programs have the same kind of user interface, i.e., the same "look and feel." A very different kind of integration would be achieved if all views in an environment worked with data files in a common format. For the purposes of this thesis, an integrated environment is one in which a dynamic collection of views can work together on a single software system so that changes made to the system by one view can be seen by other views. Designing an effective integration mechanism for environments of this type is a difficult task, because the demands made on the mechanism are stringent and varied. Some of the most important considerations are:

- **Writing New Views:** It should be easy to write new views for use in an integrated MVDE. In particular, there should be no need for the authors of a new view to understand the inner workings of each of the other views already in the environment. Similarly, there should be no need for the authors of a new view to be familiar with all the data manipulated by the other views; it should suffice for the authors to know about the data that the new view will manipulate.

- **Adding New Views:** Once a new view is written, it should be easy for its authors to add the view to an MVDE. Ideally, there should be no need to change any of the other views in the environment when the new view is added, but if changes are required, making them should be a well-defined and localized task. In addition, it should be possible to add views to an environment that is already in use — there should be no need to "shut down" the system and/or regenerate the environment from scratch.

- **Editing Multiple Views:** It should be possible for different views, possibly used by different people, to be active at the same time. Except in cases where the views are

inherently read-only (e.g., profiler output), the data presented by those views should be editable. Read-only views are useful, but read-write views are substantially more powerful.

- **Maintaining Consistency:** Views that manipulate the same data should maintain a consistent view of that data.[1] If the data is changed in one view, all other views of the data should be updated without undue delay. (The notification may not be immediate, because it may make more sense to bundle a series of low-level changes into a single higher-level modification. For example, a deletion followed by an insertion in one view may be communicated to other views as a replacement.) Important implications of changes to data should be uniformly handled by all views. For example, in an MVDE that supports dataflow analysis, all views should uniformly handle changes in a program that affect the dataflow behavior.

- **Avoiding Duplicated Effort:** There should be no need for different views to perform the same task; common operations should be directly supported by the integration mechanism. For example, in an MVDE dedicated to a particular language (e.g. C or FORTRAN), there should be no need for more than one view to parse the language. Eliminating redundancy not only reduces the amount of work needed to create a new view, it also helps maintain consistency within an MVDE by ensuring that any given task is only performed by a single agent.

  A second aspect of redundancy elimination is to stress the importance of incremental algorithms. If a view has already expended the effort to build a particular data structure, it shouldn't have to rebuild the entire structure in the face of a small change to the data. For designers of integration mechanisms in MVDEs, this means that the mechanisms should be designed to mesh well with and to take advantage of known incremental algorithms (e.g., incremental parsing and linking).

This is not an exhaustive list of the considerations that must go into the design of an integration mechanism. The mechanism should also be robust in the face of hardware or software malfunctions, it should provide concurrency control so that no data item is modified by more than one person at a time, it should be portable across a wide class of machines and operating systems, etc. Nonetheless, the factors outlined here are of fundamental importance. Their resolution may not be sufficient conditions for successful integration, but they are certainly necessary.

An additional concern is an environment's robustness in the face of *evolution* [130]. According to Lehman, an inherent property of E-type software (which includes development environments) is that it *must* evolve over time if it is to maintain the satisfaction of its users [115]. Therefore, the ability of an integration mechanism to gracefully accommodate environment evolution is an important concern. Several researchers have designed systems that specifically deal with this problem [200, 189], but in this thesis I disregard this aspect of integration. Instead, I focus my efforts on the equally compelling problems associated with integrating divergent views.

[1] Not everyone agrees with this sentiment. Finkelstein and his colleagues, for example, argue that maintaining consistency during software development is sometimes neither possible nor desirable [59].

2.2 Five Approaches to Integration

In the sections that follow, I examine five approaches to MVDE integration. The first approach represents very loose integration and corresponds to common state-of-the-practice today. I introduce it primarily as a baseline against which to measure the other, more advanced, approaches. None of the approaches is ideal, but it is illuminating to examine the tradeoffs involved as one progresses from a toolkit-based environment to one founded on a single, shared representation of the software system being developed.

Each of the approaches I describe is based on real systems, rather than being derived from abstract models of integration. Real systems, however, are only rarely a pure manifestation of a single approach (see Section 2.3). Instead, they tend to embody different features from different approaches to greater or lesser degrees. In assigning systems to categories, then, I have focused on the *mechanism* used to achieve environment integration, i.e., on the means of communication between tools, and I have generalized in novel ways when that seemed appropriate. For example, my examination of message-passing systems encompasses not only systems employing traditional message-passing mechanisms like sockets, it also includes monolithic single-process programming environments where different modules within the system communicate through function calls. In this case, I have generalized the notion of message-passing to subsume all point-to-point communication mechanisms, even if the implementation of such communication is a simple function call.

For each of the prototypical approaches to integration, I generally discuss it in terms of only a single representative implementation. Most of the approaches have been implemented in different ways by different researchers, and I provide a comprehensive survey of implementations in Section 2.4.

2.2.1 Shared File System

A traditional development environment consists of a set of independent tools, each of which typically reads and writes files and interacts with users. This organization is exemplified by most operating systems and is depicted in Figure 2-1. As shown, each tool contains its own internal representation of the data it uses, and the only way tools communicate is through the file system. For all practical purposes, the file system *is* the integration mechanism.

This simple organization exhibits some important features:

- It is easy to write new tools for this kind of environment, because there are few constraints on how the tools behave. In particular, there is no requirement that tools read and write files in a consistent manner, nor that they deal with data in a uniform way.
- It is easy to add tools to the environment, because they are completely independent.
- Many tools have already been written for this kind of environment, so traditional environments usually offer a much more sophisticated level of functionality than do environments based on other integration mechanisms. This feature is frequently overlooked, but is an important practical consideration for users.

On the other hand, this kind of integration is rife with disadvantages. There is no direct communication between tools, nor is there any safeguarding against two tools attempting

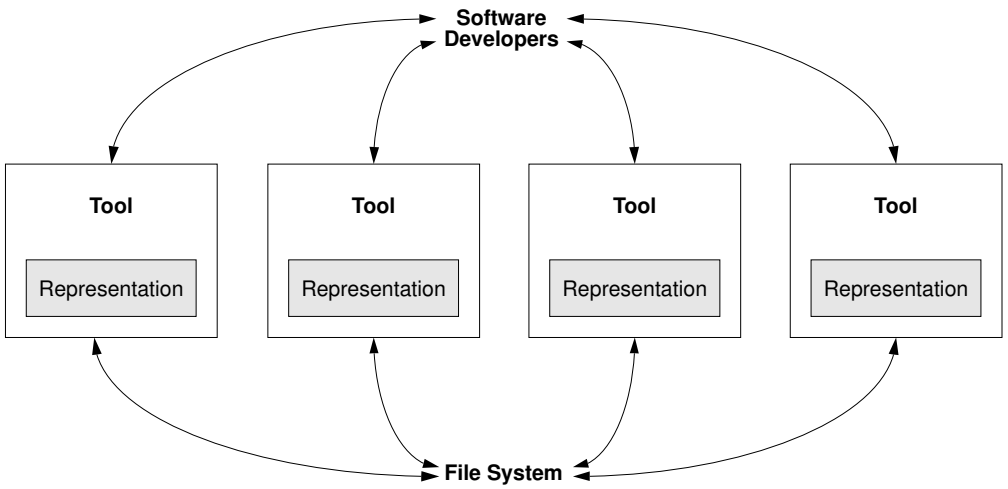


Figure 2-1: Environment integration via a shared file system.

to modify the same data at the same time.[2] As a result, it is difficult to support multiple concurrent views because it is infeasible for more than one person to work with the data in a single file at one time. The traditional mechanism also suffers in the areas of consistency and redundancy. Because each tool creates its own representation, the likelihood is high that different tools will treat a piece of data in different ways. (Even different compilers for the same programming language often contain subtle incompatibilities.) In addition, duplication of effort is common, especially in the areas of parsing and reading and writing files.

In short, integration based only on a shared file system allows new tools to be easily created and added, but offers little in the way of consistency, inter-tool communication, support for multiple developers, or elimination of redundant effort.

2.2.2 Message Passing

The FIELD environment [165] was motivated by a desire to overcome many of the drawbacks of traditional environments while still taking advantage of the plethora of tools that already run under Unix. The crux of the idea behind FIELD is that existing independent tools can be made to communicate with one another if they are modified to send and receive messages. These messages allow tools to coordinate their actions and to maintain the consistency of data that is common among them. Fundamental to FIELD is a new tool — a Message Server — that regulates the message traffic in the environment. The Message Server acts as a filter on messages to ensure that each tool receives only those messages that are of interest to it. The FIELD approach to integration is shown in Figure 2-2.

Writing new tools (or adapting existing tools) for use in FIELD is more complicated than it is for a traditional file-based environment. The complication lies in the need to write

[2] Most file systems will prevent more than one program from opening the same file for writing at one time, but this level of security fails to address the common convention whereby a program opens a file for reading, modifies the data, then writes out the modified data. Some programs (e.g., GNU Emacs [192]) attempt to prevent more than one person at a time from editing a single file, but my contention is that this is a duty for the *integration mechanism*, not something that should be left to individual tools in the environment.

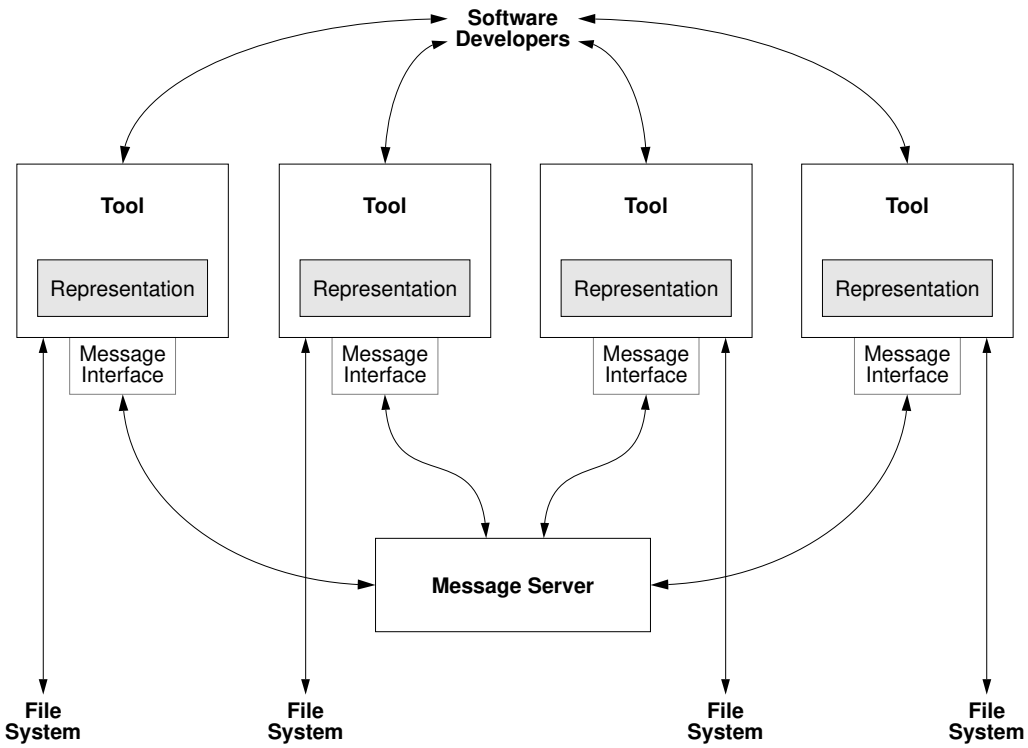


Figure 2-2: Environment integration via message passing.

a message interface to allow communication with the other tools in the environment. This interface performs three functions. First, it allows a tool to register a set of message patterns with the Message Server; the server then ensures that only messages that match a registered pattern are forwarded to the tool. This technique is known as *selective broadcasting*. Second, it allows a tool to send messages to the server for selective broadcast to other tools. Third, it allows a tool to receive messages of interest from the server; the tool can then process those messages in whatever way it sees fit.

Writing a suitable message interface can be a tricky business. Before the authors of a tool can specify patterns for the messages of interest, they must find out which messages exist and what information they convey. FIELD provides no authoritative index of messages, so tool authors must examine the existing tools in the environment to discover which messages are generated, their content, and format. The impact of this requirement is amplified by the fact that it is an ongoing concern: as tools are dynamically added to or removed from FIELD, the set of possible messages can change.

When tool authors choose the set of messages they will generate, they must attempt to predict which of the tool's actions and/or modifications to data might be of interest to current or future tools. This is a daunting enough task in and of itself, but in FIELD there is a subtle problem that exacerbates matters. The problem is that the tools in a FIELD-like environment lack a common semantic language — even a set of common semantic concepts — that can serve as a foundation for effective communication. For example, an editor might want to send and receive messages about lines of a file, but a compiler might prefer messages that deal with identifiers and functions; a profiler might want to converse in terms

of program counters, etc. Given tools with such varied perspectives, it is difficult to imagine how the simple exchange of messages can fully integrate their actions.

When adding a new tool to FIELD, it may be necessary to modify existing tools to take advantage of the information in the messages the new tool provides. If the information is not needed by any tool, the new messages can safely be ignored — no decrease in integration occurs (although a possibility for increased integration is left unexploited). However, if the information is currently generated internally by an existing tool, it is important to update the existing tool to use the new messages; otherwise there is redundant computation in the environment, and the specter of inconsistency arises.

Like the tools in a traditional environment, tools in FIELD maintain their own representation of the data they manipulate. Hence they tend to exhibit the same types of redundant computations and inconsistent data manipulations that are found in traditional tools. For example, both a program pretty-printer and a compiler would have to expend the effort to parse a source file, and it is unlikely that the results of the parse would be completely consistent with one another. It is possible to employ messages to mitigate problems of this nature, but the problems cannot be eliminated using that mechanism alone.

In spite of these limitations, experiments with FIELD have proven remarkably successful. Using selective broadcasting as an integration mechanism, FIELD has been expanded to encompass interfaces to over a dozen different tools, including an editor, compilers, debuggers, an interactive program for displaying and modifying dynamic data structures, a program profiler, and many others. The FIELD approach to systems integration is also noteworthy for its openness. Compared to the integration mechanisms discussed in the next three sections, it is relatively easy to modify existing Unix tools to participate in FIELD.

FIELD offers clear advantages over traditional environments, and it has been shown to be a practical approach to integration, but important open questions remain. Steven P. Reiss, the creator of FIELD, has so far authored almost all the tools in the environment, and it is still unclear how well an integration mechanism that works well for a single developer will scale to work well with multiple developers. This concern is especially significant in view of FIELD's lack of a formal management scheme for message names, content, and format.[3]

It is possible to manage the complexity of arbitrary messages, senders, and receivers by eliminating much of the flexibility of that approach. In fact, it is possible to eliminate all the runtime flexibility by fixing at compile-time which messages are sent from which tools to which other tools. Communication between pairs of tools is then achieved not by dynamically processed messages, but by mutual function calls, as is shown in Figure 2-3.

This restricted (static) form of message passing has certain advantages. For example, when writing a function to map data between two tools, the author of the function need only understand the two views in isolation — other tools in the environment are unaffected. Because only two tools need be considered, custom tool-specific data structures can be easily accommodated. As a result, for a small, fixed set of tools, this approach — pairwise

[3]Several commercial development environments employing the FIELD approach to integration are now available, and an IEEE technical committee (X3H6, "CASE Tool Integration Models") has been established to try to standardize the syntax and semantics of the most common messages used in such environments. Independent of this committee, two different message standards have been published [42, 36]. The proponents of these documents are working with X3H6, but it is difficult to avoid recalling the maxim that the nice thing about standards is that there are so many to choose from.

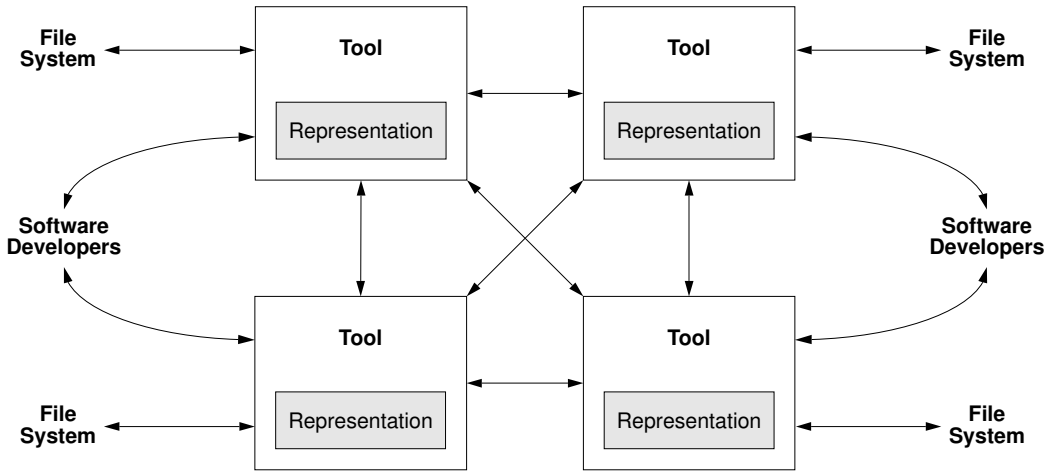


Figure 2-3: Environment integration via pairwise mappings.

mappings between tools — may actually be the easiest approach to integration.

Nonetheless, there are serious disadvantages to pairwise mappings. First the number of mappings grows quadratically with the total number of tools, so maintaining a high level of integration becomes impossibly difficult as the number of tools increases. Second, a change to a single tool may well necessitate updating all the other tools in the environment. Finally, there is still no guarantee that a tool will communicate with other tools when it makes a significant change to the system under development. Like FIELD-based message passing, an environment built on pairwise mappings is still hostage to the voluntary cooperation of the tools in the environment.

2.2.3 Simple Database

One way to increase the integration in a traditional environment is to move the data from a simple file system into a more formal database. Within a typically rigid framework (e.g., relations), databases provide great flexibility in the data that can be stored, and their support for locking and transactions provides a firm foundation on which to build an environment in which data can be safely manipulated by multiple developers working concurrently. A number of environments have been built around databases (see Table 2-3). A database-centered architecture for an environment is shown in Figure 2-4.

Although tools may share a common database, the data structures supported by the database are generally not sophisticated enough to be directly used by the tools in the environment. Even in databases supporting complex data structures, tools tend to operate by first retrieving all the data they need from the database; then building a custom-tailored internal representation, which they manipulate as the program runs; and finally writing out the modified data all at once. Because of this convention, communication between tools sharing a database is often no better than in traditional environments; a tool that is running usually has no idea what is being done by the other tools that are running at the same time. Exacerbating this problem is the fact that tools in an environment often lock data for fairly long periods of time, thus preventing other tools from seeing the data during the course of the transaction. Finding ways to cope with the implications of such

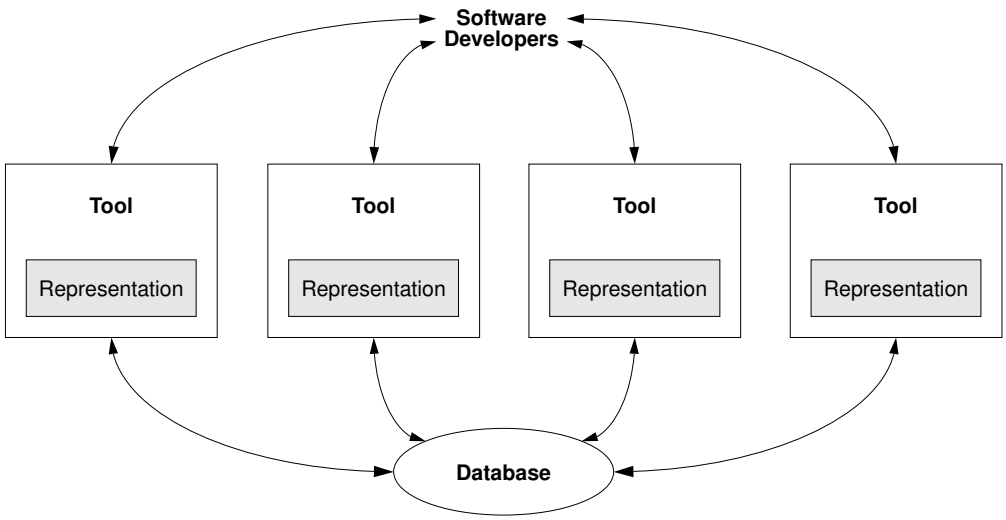


Figure 2-4: Environment integration via a simple database.

long-duration transactions is an active area of database research [228].

Some databases allow users to define *triggers*, i.e., events in a database that cause certain actions to take place, and this could form the basis for some communication between tools. However, because most events of interest occur in a tool's internal data representation instead of in the database, this facility is of limited utility. Furthermore, databases are often updated in relatively course-grained increments (e.g., a complete function might be the unit of incremental update), and this limits the resolution of the modifications to data that can be communicated.

Databases offer good opportunities for the automatic maintenance of data consistency, not only because of their support for locking, but because they may allow for the specification of global constraints that are automatically satisfied by the DBMS. For example, database constraints could be used to perform a function similar to the Unix make program, thus ensuring that the executable version of a system is constructed from the most recent sources. This is in fact the underlying basis for Odin's Derivative Forest [41], which is a generalized mechanism for indicating how objects are produced from one another.

Writing tools for use in an environment founded on a database is complicated by the fact that it is often necessary to understand all of the data in a database in order to write the tool. In particular, authors of new tools must take care to ensure that they do not create any new database structures that duplicate data that is stored elsewhere. Alternatively, they may choose to deliberately duplicate data, but then they must carefully arrange for the different copies of the data to be maintained consistently. Either way, they must have a deep understanding of the database in its entirety. Unfortunately, the addition to, removal from, and modification of tools in the environment conspire to keep the database in a state of flux, so it is a difficult task to retain one's understanding of it. This problem becomes especially acute as the database grows larger through the addition of new tools. It eventually becomes difficult to modify the database at all, simply because it is so complex that it defies comprehension by most potential users.

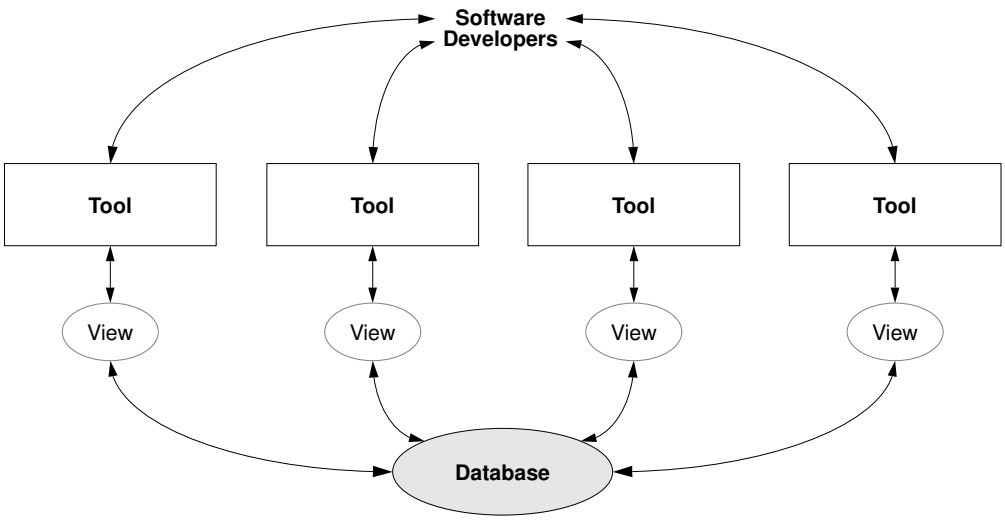


Figure 2-5: Environment integration via a view-oriented database.

2.2.4 View-Oriented Database

Simple databases allow tools to create custom representations of system data, but they fail to facilitate communication between tools. This communication problem would be solved if all tools used the database representation for system data, but then tools might be forced to use data structures that were not well suited to the operations they had to perform. David Garlan studied this problem, and his solution to the impasse was to design a mechanism whereby different tools use different representations ("views") for data in the database, but data shared between views is automatically kept consistent by the DBMS [65, 66]. In Garlan's words, "a view is a virtual description of the common database, defined in such a way that objects can be shared among a collection of tools, each tool accessing objects through the views it defines. The common database then becomes the *synthesis* of all of the views defined by the tools in the environment." The view-oriented approach is depicted in Figure 2-5.

Tools in an environment typically use more than one data structure, so tools using a view-oriented database bundle sets of associated views into *features*. Features are similar to Ada packages and Modula-2 modules in that they explicitly import the features they use, and they construct and explicitly export the views they implement. They are also the primary mechanism for sharing data structures between tools. One of the fundamental tasks in developing tools for this kind of environment, then, is to identify the relevant views and features that have already been developed so that these views and features may be reused (and the data in them shared). In this respect, creating tools for a database with views is not dissimilar to creating them for a simple database system. The aggregation of views into features eases the problem of finding out which data already exists, but as the number of views and features increases, it becomes more and more difficult to comprehend the database as a whole.

At the core of the view-oriented database is the mechanism that allows data to be shared between views: the *compatibility map*. Compatibility maps describe how operations on data in one view translate into operations on the same data in another view. For example, a

collection of objects might be viewed as a set or as an ordered list. Appending two ordered lists would map to a set union, and reordering a list would map to a null set operation. By defining compatibility maps between arbitrary types, users can achieve arbitrary degrees of data sharing, with the system automatically maintaining consistency between views of the data.[4] Unfortunately, it can be difficult to create compatibility maps, especially for views that are dissimilar. As a result, writing the compatibility maps necessary to add a new tool to an existing environment can be an arduous, time-consuming task.

As designed by Garlan, a set of views, features, and compatibility maps collectively define an environment database system, and each system is custom-generated for the environment it is to support. When new views, features, or maps are to be added, the database system must be regenerated. Since adding a tool to an environment virtually always means adding a new view, Garlan's implementation calls for the environment to be regenerated each time a new tool is added. It would probably be possible to eliminate this database compilation step by interpreting the compatibility maps and the dynamic views (i.e., database constraints), but that strategy would exact a performance cost at runtime.

A view-oriented database offers a number of advantages. It has all of the strengths of a simple database, it offers tools the ability to manipulate data using the data structures they find most convenient, and it automatically maintains shared data in a consistent state. Its primary drawbacks are that it can be difficult to construct tools for this kind of environment, and it can also be complicated to add tools to an environment centered around a database with views.

2.2.5 Canonical Representation

Many problems with the database approaches described in the previous sections would be solved if all the tools in an environment shared a single representation that was stored in the database. Unfortunately, no representation has yet been identified that is suitable for all possible tools. Nonetheless, the idea of having all tools operate on the same data structures is an attractive one, and a number of environments have been built around canonical program representations. Figure 2-6 depicts this organization.

The allure of a canonical representation — in some sense the Holy Grail of environment integration — is a result of the many advantages that would accrue from its use. With only a single representation for system data, tools communicate directly and with arbitrarily fine resolution. The integration mechanism itself can assume the responsibility for common operations, including, significantly, that of *change notification*. Like view-oriented databases, then, and unlike the other integration mechanisms discussed in this thesis, an environment based on a canonical representation is not dependent on its constituent tools' voluntary cooperation in notifying other tools of changes.

Because the primary data structure is shared in a canonical representation, data consistency is not an issue. The task of creating new tools is simplified by the fact that only a single (albeit complex) data structure need be understood, and adding tools to the environment is streamlined by the fact that all tools interact with the data and with one another in a uniform manner. A suitably chosen representation can allow the environment to take

[4] Garlan's original compatibility maps were actually somewhat more restrictive, but followup work by Habermann and his colleagues [77] has extended them to this level of generality.

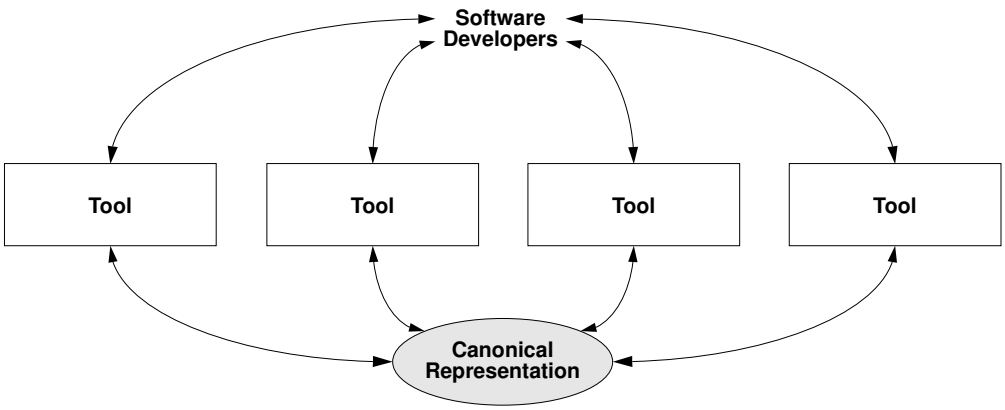


Figure 2-6: Environment integration via a canonical representation.

advantage of incremental algorithms for parsing, code generation, dataflow analysis, and the like [82].

Designers of canonical representations typically choose some fundamental data structure for the "core" data, and then allow tool-specific data to be added to the structure in some manner. For example, almost all language-specific programming environments employ an abstract syntax tree (AST) as the core data structure, and then allow tool-specific "decorations" of the tree, e.g., annotations or attribute equations. (I use the generic term "annotation" to refer to such tool-specific data that is somehow attached to the core data.) One of the advantages of this organization is that the fundamental data structure (e.g., the AST) bounds the amount of information that must be mastered by the potential creator of a new tool. This is in contrast to the database-centered environments I have already examined, where there is no structural differentiation between core data and tool-specific data. As a result, it is easier to write tools for environments with canonical representations than for environments employing more general-purpose databases. On the other hand, fully integrating a new tool with existing tools still requires that the authors of the new tool understand both the core structure as well as the annotations, since there may be data in the annotations that should be shared.

Experience with AST-based environments has revealed that ASTs are not a flexible enough representation to support a sufficiently wide class of tools. In fact, Garlan's work on view-oriented databases was motivated by difficulties encountered when trying to expand the AST-based Gandalf environment [78]. Similarly, Reiss' experience with PECAN [163] led him to conclude that ASTs are particularly poorly suited for nonlinear views of programs, especially graphical views. I suspect that any representation based on syntax will prove to be too limiting, including those based on abstract syntax graphs.

In some sense, a canonical representation is a simple database pushed to the limit — to the point where the data structures are rich enough to be used directly by tools. In that case, the description of the canonical representation becomes the schema for the database. Finding ways to specify such schemata and to efficiently implement them is an active area of database research, and is one of the primary motivational forces behind research into object-oriented databases [91, 56].

Criterion	Shared File System	Message Passing	Simple DB	View-Oriented DB	Canonical Rep.
Writing New Tools	Good	Fair	Poor	Poor	Fair
Adding New Tools	Good	Fair	Poor	Poor	Fair
Simultaneous Views	Poor	Fair	Poor	Good	Good
Consistency Maintenance	Poor	Fair	Good	Good	Good
Redundancy Avoidance	Poor	Poor	Fair	Good	Good

Table 2-1: A comparison of the five approaches to environment integration.

2.3 Discussion

These are not the only ways to approach environment integration, of course, nor are these mechanisms necessarily exclusive. Most real systems are in fact hybrid approaches, in which two or more "pure" approaches are combined in an attempt to develop a system with the advantages of all of its components, but with fewer disadvantages. For example, FIELD can be considered a combination of a pure shared file system with a message passing system. For FIELD, selective broadcasting enables direct communication between applications, while the shared file system facilitates the use of existing applications. For tools designed to work with a common database, similar leverage of that tool base could be achieved by combining message passing with a simple database.

Table 2-1 provides a very coarse-grained summary of the relative strengths and weaknesses of the five approaches discussed in this chapter. The table lists how much support the different approaches provide for the five factors I introduced earlier: writing new tools; adding new tools; using multiple simultaneous views; maintaining consistency in the environment; and avoiding redundant code and computation.

The contrast between the characteristics of the shared file system environment and those of the view-oriented database environment is striking. It suggests a fundamental tradeoff in integration mechanisms: one can either have an environment in which it is easy to develop new tools, or one can have an environment in which the tools communicate well with one another, but it seems to be quite difficult to achieve both. It is not surprising that it requires less effort to develop standalone tools than tools that cooperate closely with other tools, but it is disheartening that *so much* additional effort seems to be required.

A significant contributor to this additional complexity is the fact that authors of cooperating tools must devote themselves to understanding what data the other tools manipulate and how they manipulate it. At present, we appear to lack adequate mechanisms for effectively managing this complexity. Garlan's strategy of grouping related views into features was an important step toward imposing useful abstraction on a database, but the difficulty involved in creating new compatibility maps for his view-oriented databases detracts from the overall practicality of his approach.

FIELD achieves an impressive level of integration within what is effectively a traditional

development environment. Its success in this arena is a consequence of the fact that FIELD directly confronts the problem of traditional tools' not communicating with one another. By providing support for such communication, FIELD allows tools to share the results of computations at runtime. Shared computations allow tools to avoid redundant effort, and also allow them to maintain consistency in the data they share.

Fundamentally, however, FIELD is nothing more than a message clearinghouse, and this limits the services that it can perform. It can never ensure that tools send out the messages needed by other tools, nor can it guarantee that tools will react consistently to messages they receive. In short, FIELD is a success only as long as the tools in the environment voluntarily cooperate, but it cannot coerce recalcitrant tools (or tool writers) into participating in the integration of an environment. This is a very different situation from environments based on a view-oriented database or canonical representation. In those environments, the integration mechanism can guarantee a certain amount of cohesiveness between tools simply because the database or representation maintains control over the data manipulated by the tools. Like it or not, tools in these environments make their actions public every time they modify the common data structure, and the integration mechanism has the authority to react accordingly. Systems based on message passing simply don't have that kind of power.

2.4 A Categorization of Existing Systems

One of the contributions of this thesis is the insight that virtually all existing work on MVDEs employs as its primary approach to integration one of the mechanisms described in the foregoing sections. A related contribution is my categorization of environments described in the research literature on the basis of the primary integration mechanism each uses. This categorization is summarized in Tables 2-2, 2-3, and 2-4. Each entry in these tables is the name of the system, if available; the name of the specific integration mechanism, if no system was named in the published report(s); or the names of the researchers, if neither a system name nor an integration mechanism name was published. Within a category, environments are listed in alphabetical order by system name, integration mechanism name, or researchers' names.[5]

Table 2-2 lists multiparadigm environments (see below), Table 2-3 lists MVDEs that use an integration mechanism other than a canonical representation, while Table 2-4 lists MVDEs that do use a canonical representation. It should be noted that many environments take a particular data structure and put this data structure in a database to provide persistence and/or concurrency control. Such environments are listed as using the particular data structure rather than as using a database, because the data structure has more to do with supporting integration than does the database. As noted earlier, most environments are hybrid architectures, but the organization of the tables reflects the important underlying similarities for the environments that are grouped together.

[5]It might have been more interesting and useful to organize environments within a category chronologically, but there is often a discrepancy between the date of first appearance of a system and the date of publication of a good reference, and I have tried to limit my citations to accessible and comprehensive references.

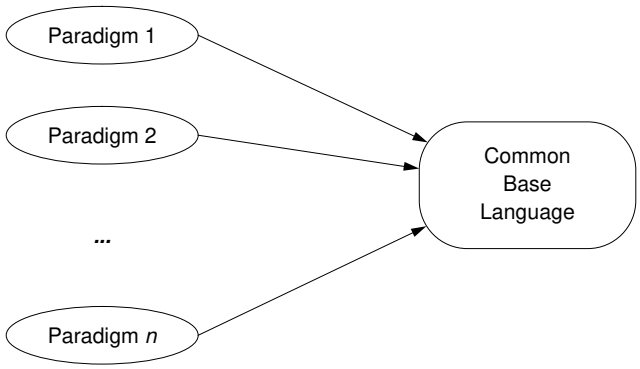


Figure 2-7: A common architecture for multiparadigm environments.

2.4.1 Multiparadigm Development Environments

My definition of an MVDE requires that views of a system be able to communicate with one another in some way as the system is edited. This definition excludes an interesting class of environments, namely *multiparadigm* environments. It is worth a brief digression to discuss such environments now.

Multiparadigm environments can be considered MVDEs in which the views do not interact during editing. In such an environment, different parts of a single system may be developed using different languages or language paradigms, and the system as a whole is simply the union of its constituent parts.

The implementation of a multiparadigm development environment almost always employs one of two general architectures. The most common architecture is for each view to be mapped into some common underlying base language. When such a multiparadigm program is run, it is the common base language which is actually executed. This architecture is shown in Figure 2-7. There is typically no back-mapping from the common language to the "source code" which generated it, or, at best, each portion of the common language can only be mapped back into the paradigm from which it was generated. Multiparadigm software development based on this architecture is thus akin to composing a program from routines written in, say, FORTRAN, C, and Ada, and then compiling everything into a common assembly language. Each language is in some sense a view, but the views do not overlap.

An alternative architecture is to work with each paradigm independently, ultimately generating a separate executable version of each part of the system. At runtime, the different parts of the system execute as separate processes, communicating via remote procedure calls, sockets, or some other mechanism for interprocess communication. Using this architecture, there never is a common representation of the system as a whole. Like the architecture founded on a common base language, there is typically no way to map from one paradigm to another. Integration across views is achieved only at runtime.

The research into multiparadigm environments is summarized in Table 2-2. The Draco system [140, 141], Ipser's subsequent work on multi-formalism specification environments [98, 99], and Wile's research into "local formalisms" [221] explicitly provides support for (textual) user-defined languages that translate into a common base language, while Garden [164] and the Visual Programmer's WorkBench [179] concentrate on the creation of

Approach to Integration	System (Researchers)
Underlying Base Language	(Agha [1]) CIP-L [17] Daisy II [62] Draco [141, 140] G-2 [157] Garden [164] (Golin and Reiss [71]) (Ipser [98, 99]) Leda [33] Loops [22, 195] PLEASE [209] V [188] Visual Programmer's WorkBench [179] (Wile [221]) (Zave and Jackson [227])
Communicating Processes	CODE [32] (Koschmann and Evens [111]) (Zave [226])

Table 2-2: Multiparadigm environments.

multiparadigm visual languages. Other multiparadigm work includes Zave's research into composing different independent paradigms [226]; the explicitly multiparadigm languages Loops [22, 195], Daisy II [62], Leda [33], and G-2 [157]; Koschmann's and Evens' research on mixing object-oriented and logic programming [111]; Agha's proposal to use Actors as a basis for concurrent programming [1]; the CODE system for developing parallel applications [32]; the wide-spectrum languages V [188], CIP-L [17], and PLEASE [209]; and the work by Zave and Jackson on formally composing partial specifications [227].

2.4.2 MVDEs Not Using a Canonical Representation

Table 2-3 summarizes research into MVDEs that achieve integration by a mechanism other than a shared canonical representation. The table excludes traditional environments based on a shared file system (e.g., Unix), as these traditional environments are, as I remarked earlier, the baseline against which all other work is measured.

Environments based on message passing fall into two basic camps. Environments following the FIELD model of Section 2.2.2 include, in addition to FIELD itself, the commercial implementations SOFTBENCH [34], FUSE [51], ToolTalk [202], and CASEVision/Workshop,[6] as well as the research efforts Forest [67], ISM [176], and the work on Mediators [200]. Forest extends the FIELD model by adding support for *policies*, which offer flexible user-defined control over the conditions under which messages are passed. ISM employs a message-based approach to integration, but it has more of an AI perspective to its application. Significantly, ISM doesn't seem to consider the possibility that one message might have multiple recipients. Sullivan's and Notkin's goal in their work on Mediators is a generalization of message-based integration that is designed to better support environment

[6] CASEVision/Workshop is a commercial product from Silicon Graphics, Incorporated.

Approach to Integration	System [Mechanism] (Researchers)
Message Passing	CASEVision/Workshop Cedar [205] FIELD [165] Forest [67] FUSE [51] Interlisp [206] ISM [176] [Mediators [200]] MLP [85, 84] [PCTE IPC [25, 57]] Prisma [142] Smalltalk [210] SOFTBENCH [34] [ToolTalk [202]] [ViewPoints [145, 60]]
Simple Database	Amadeus [21] CIA [39] CIA++ [73] COPE [7] Faust [76] MicroScope [6] Odin [41] OMEGA [121] [PCTE OMS [64]] PDS [38] RPDE^3 [81] Trellis [147] XREFDB [118]
View-Oriented Database	(Garlan [65, 66]) Janus [77]

Table 2-3: MVDEs not based on a canonical representation.

evolution. Evolution is an important topic, but, as noted in Section 2.1, one not directly addressed by my work in this thesis.

The common feature of FIELD-like environments is the loose coupling between the views generating messages and the recipient(s) of those messages. Message senders need not explicitly state who is to receive a message, and in fact the set of recipients is typically determined dynamically. A very different message-based architecture is predicated on sets of hardwired mappings between views. Examples of such systems include Prisma [142] and ViewPoints [145, 60]. Each of these systems supports multiple independent views, but each also relies on environment developers to write bidirectional mappings between views that are to be kept consistent. A more familiar example of such a tightly coupled architecture is exhibited by closed, monolithic (typically everything is in a single address space), tightly-integrated, single-language environments like Interlisp [206], Cedar [205], and Smalltalk [210]. In these environments, seemingly independent views are not really independent, because they are inextricably dependent on the other views in the system. For

example, the Interlisp `File` package added a capability to notice when data was changed by defining a new function, `MarkAsChanged`, and then inserting calls to `MarkAsChanged` "in all of the parts of the Interlisp system that changed objects — the editor, the DEFINE function, the facility for (re)declaring records, and DWIM" [206, p. 27].

Another message-based architecture that may seem to support multiple views is embraced by algorithm animation systems based on "interesting events." Some of the best known of such systems are Balsa [29, 30], TANGO [193, 194], and Voyeur [191], but there is a fundamental difference between these kinds of systems and the MVDEs that are the subject of my work: systems based on interesting events cannot provide views of a program until the program is instrumented to produce the interesting events. For my work, a view of a system is of no interest unless it can be generated without requiring modifications to the system being worked on.[7,8]

Several environments have been developed that store information about programs in a relational database, e.g., OMEGA [121], CIA and CIA++ [39, 73], and XREFDB [118]. In each case, however, the goal of the effort has been to answer queries about static programs, not to provide communication between views of dynamically changing systems. For example, each of the above systems uses a batch process to load the database; the ability to make incremental changes was not a primary design consideration. In addition, none of these systems supports the execution of a program in its representational form.

Environments based on simple databases typically break systems under development down into "natural" syntactic chunks (such as modules, functions, and variables) and "natural" semantic chunks (such as symbol tables and cross-reference listings). These chunks are then stored in whatever form the database supports, e.g., relations in the case of OMEGA, objects in the case of MicroScope [6]. Some database-centered environments, such as Trellis [147] and Odin [41], use larger-grained chunks, possibly as large as entire files of source and object code.

With the exception of minor extensions to compatibility maps by his erstwhile colleagues [77], Garlan's work on view-oriented databases has attracted little follow-up work by the research community.

2.4.3 MVDEs Using a Canonical Representation

Table 2-4 summarizes MVDEs that achieve integration by means of a shared canonical representation. As noted in Section 2.2.5, the most common representation investigated has been the abstract syntax tree (AST). ASTs are at the core of many well-known programming environments, including the Cornell Program Synthesizer [168], Gandalf [78], Pecan [163], R^n [89], MENTOR [53], and CENTAUR [24]. 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

[7] If interesting events could be automatically generated for a program, animation views would quite interesting indeed. Whether it is possible for interesting events to be automatically inserted in programs, however, is still an open question. To date, there are no systems that do it.

[8] A different outlook on the matter leads to the conclusion that MVDEs and systems for algorithm animation are actually close cousins. Systems that animate algorithms generally help users visualize *data*, whereas MVDEs generally help users visualize *programs*. These are simply two sides of the same coin, because programs *are* the data visualized by MVDEs. Nonetheless, the practical problem of coming up with the interesting events that drive an animation should not be underestimated.

Canonical Representation	System [Mechanism] (Researchers)
Abstract Syntax Tree	Centaur [24] Cornell Program Synthesizer [204] Fortran Programming Environment [89] Gandalf [78] Magpie [49] MENTOR [53] MultiView [5] Pan [11] PECAN [163] Poe [61] Syned [90] Synthesizer Generator [169, 168] (Tenma *et al.* [207])
Abstract Syntax Graph	IPSEN [139, 120] Maintainer's Assistant [158] TEAM [40]
Plan Calculus	Cobbler [220] KBEmacs [219] Recognizer [174] Satch [220]
Miscellaneous	[Attributed Graph Specifications [3]] [General Design Representation [123]] (Lea [112]) [Program Dependence Graph [151]] [Typed Hypermedia [43, 14]] [Unified Interprocedural Graph [82]]

Table 2-4: MVDEs based on a canonical representation.

representation for these syntax-based views. However, 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 the environments mentioned above allows programmers to define new views. Similar difficulties arise with program representations based on abstract syntax graphs, such as those used in IPSEN [139], the Maintainer's Assistant [158] and TEAM [40].

The Programmer's Apprentice project [172, 173] has employed the Plan Calculus [170] for a variety of systems. The Plan Calculus is a graph-based representation for programs that incorporates both control- and dataflow information. It has been successfully incorporated into the KBEmacs system for partial program synthesis [219], into Satch and Cobbler for automatic program translation [220], and into the Recognizer [174], a program that identifies design features in existing programs. Rich and Waters, the primary architects of the Programmer's Apprentice project, suggest that the Plan Calculus would be well-suited to supporting multiple interacting views, but they have so far chosen not to pursue this avenue of research [172].

Program dependence graphs [58] were originally developed as a way to facilitate code optimization and/or parallelization, but since then they have found favor in a variety of

applications, including interactive code restructuring [74] and integration of program variants [93]. Ottenstein and Ottenstein have proposed that PDGs could be used as a basis for an integrated development environment [151], but the form of a PDG is very different from the views that programmers typically use, 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 incremental algorithms for common operations in an interactive environment, such as adding a statement to a program under construction. There are many derivatives of PDGs, including program representation graphs [224], program dependence webs [12], system dependence graphs for interprocedural slicing [94], and unified interprocedural graphs [82].

Research into hypermedia representations of system information has resulted in mechanisms for storing data that is similar to SPGs. For example, the Hypertext Abstract Machine (HAM) [35] used in Neptune [50] has a resemblance to SPGs. The HAM offers graphs, contexts, nodes, links, and attributes; contexts are used to partition data in graphs, and their functionality can be realized using node/link attributes and filtering predicates. Significantly — and unlike SPGs — the hypergraph itself has no semantics. Instead, attributes are used to assign semantics to the nodes and links. Neptune is designed for use as a CASE environment, and as such tends to deal with things at a fairly high level of abstraction. For example, in discussions of source code, it refers to a single canonical textual representation (e.g., Modula-2) and the finest granularity it seems to consider is that of function definitions.

A variety of other canonical representations has been proposed or implemented for software development environments. Lubar's General Design Representation [123] is used in the ROSE-2 [124] environment, but the GDR is primarily designed to represent static design data, and has not addressed issues concerning the executability of the resulting representation. Lea has proposed an internal representation for C++ and other object-oriented languages [112], but, much like assembly language output from compilers, his representation is not designed to support back-mapping to a "source" view once the representation is constructed.

2.5 Conclusions

At the outset of this chapter, I defined five important criteria for view integration in an MVDE, and in Table 2-1 I compared the effectiveness of five fundamental integration mechanisms in terms of these criteria. Although none of the mechanisms was found to be ideal, the most promising mechanism was the canonical representation. Among its most attractive strengths are:

- **Adding new tools to an environment typically does not require changes to existing tools.** Because tools base their activities only on a (controlled) shared data structure, adding a new tool to an environment has an effect on other tools only indirectly (through changes the tool makes to the canonical representation).
- **Change notification can be handled directly by the integration mechanism**; tools *must* make visible to the integration mechanism the changes they make to shared data. Similarly, the integration mechanism is a natural location for the implementation of other tool-independent functionality in the environment.

It is these strengths that have drawn so many researchers to the canonical representation, a level of interest that is manifested by the entries in Table 2-4.

Unfortunately, the research to date on canonical representations has not had as its goal the support of MVDEs with widely varying and interacting views, much less user-definable views. Instead, canonical representations have been developed primarily to support environments centered around a single primary view and some fixed set of subsidiary views. Within such environments, the primary view is *the* program source code, so only that view may be edited. Interactions between views is unidirectional: changes flow *from* the primary view *to* the subsidiary views. The representation itself is carefully crafted to support the views that are known to exist; no consideration is given to the possibility that developers may wish to create new views. In short, the research on canonical representations has focused on closed environments with a single distinguished "source code" view and some fixed set of read-only subsidiary views.

It is hardly surprising, then, that existing representations fall short of what is needed for truly effective MVDEs. What is called for is a new form of canonical representation, one that is custom-tailored for the particular needs of an MVDE. In Chapter 3, I examine these particular needs in detail, and in Chapter 4, I introduce a representation that meets these requirements: Semantic Program Graphs.

Chapter 3

Designing a Representation

I established in Chapter 2 that the most appropriate integration mechanism for an interactive MVDE is a canonical representation (CR) of the system under development. I now turn my attention to the factors that constrain the design of a suitable canonical structure.

A good CR must be able to support the entire range of views that one wishes to offer in an MVDE. Ideally, this range would be unconstrained – one would like to allow all conceivable views — but, as I show in Section 3.1, that is impossible, even in principle. One must therefore settle on support for some subset of the universe of imaginable views. But how is this subset to be specified?

I am interested in MVDEs offering an open-ended set of views, so it is not possible to simply list the languages (views) that are worthy of support. Instead, I describe the different *paradigms* of computing I wish to support, and I identify the characteristic features of each paradigm. The union of these features comprises a constraint on the *expressiveness* of a canonical representation, because a candidate CR that cannot express these essential features is *a priori* unacceptable.

The set of features that make up this expressiveness constraint constitute what is in essence a *model* of views of software systems. If the features of a particular view V are present in the model, V can be fully supported in an MVDE built around a CR corresponding to the model. If not, then only those features of V that are also present in the model can be guaranteed support. Thus, the model provides a yardstick by which to measure how well a particular view fits into the MVDEs with which I concern myself in this thesis. I develop this model in Section 3.3.

The expressiveness constraint is a reflection of the need to be able to map from views to the CR without losing essential information. It is a necessary constraint, but its satisfaction is not sufficient to ensure that a CR is appropriate for an interactive MVDE. The inverse mapping — from the CR to views — is equally important, and it gives rise to its own constraint: that of *generalized invertibility.*

An invertible mapping is just one that can be reversed, so if a view V can be mapped into a CR, the mapping is invertible if it is possible to recover V from the CR. If we think of the mapping from V to CR as analogous to that that performed on source code by a compiler or an assembler, the inverse mapping process is analogous to that performed on object code by a decompiler or a disassembler.

In an MVDE, however, this particular kind of invertibility — from V to a CR and then back to V — is not generally the primary challenge. Rather, the concern is with the

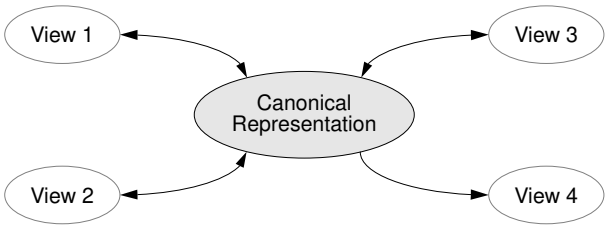


Figure 3-1: A general architecture for an MVDE based on a canonical representation. Views 1, 2, and 3 are read-write; view 4 is read-only.

ability to generate some view V_2 from the information in a CR when that information was generated from some other view V_1, and *to do it in such a way that V_2 appears more or less as it would had the software system represented by CR been developed using V_2 in the first place*. This is a very strong constraint. Among other things, it calls for the ability to (more or less) translate common idiomatic expressions in one language into their idiomatic counterparts in other languages.

Accomplishing that feat calls for great care in the design of the features of a CR, and in Section 3.5 I describe my approach to this design problem. Fundamentally, the process consists of identifying common pan-language programming concepts and then finding a way to represent them such that essentially similar concepts are represented in the same way, while essentially dissimilar concepts are *not* represented in the same way. As examples, I describe four representation problems in detail: iterative looping constructs, subprograms and calls, scopes and name bindings, and mechanisms for interprocess communication.

I conclude this chapter by revisiting the canonical representations I described at the end of Chapter 2. My decision to develop a new CR for MVDEs can scarcely be defended unless existing representations have been shown to be inadequate for the task, and in Section 3.7 I examine existing CRs in view of the criteria of expressiveness and generalized invertibility that constrain the design of a canonical representation.

3.1 The Limits of Mapping Between Views

A general architecture for an MVDE based on a canonical representation is shown in Figure 3-1; a much more detailed model is provided in Chapter 6. As shown in the figure, mappings between views and the representation are bidirectional, so a software system can be seen and edited through any view, and edits through one view are propagated to other views through changes in the CR. This is the most general case. It is also possible that some of the mappings are unidirectional; this yields read-only views. For example, the mapping between the CR and View 4 in Figure 3-1 indicates that View 4 is a read-only view. In any case, it is likely that many of the mappings from the representation to the views will be partial, because most views show only a proper subset of the information available about the software system under development.

Significantly, it is impossible to support all conceivable views. This is because, in the most general case, mapping between arbitrary pairs of views requires being able to solve the Halting Problem. Consider a software system S and a view V_1 of S that is some conventional programming language, say, Pascal. That is, V_1 is S as a Pascal program.

Consider then a second view V_2 of S that manifests itself as either the word "Halts" or the words "Doesn't Halt," where the manifestation of V_2 indicates whether S will halt when given some specified input data. An MVDE that fully supported these two views would have to be able to solve the Halting Problem for an arbitrary Pascal program, and it would have to be able to update this solution as the Pascal program was edited. It would also have to be able to make the appropriate modifications to the Pascal view as the Halting Problem view was edited. Clearly, such an MVDE cannot exist.

3.2 Paradigms For Software Development

Given that an MVDE cannot support all possible views, a critical question arises: what class of views *should* an MVDE support? The answer to this question is determined more by pragmatic considerations than by technical constraints. Because the most commonly used languages for large-scale software development are high-level, "third generation" languages like FORTRAN, C, and Ada,[1] the most commonly used views in practice are these languages themselves or are views derived from them, including call graphs, module interconnection diagrams, high-level dataflow diagrams based on functional decomposition, etc. It is most worthwhile, then, to focus on designing a representation to support this broad class of views.

Of course, this is but an informal description of the views of software systems that an MVDE should support. Making the description more rigorous requires a detailed examination of the kinds of views that should be supported. I do this in terms of *paradigms* of software development, which group together broad classes of views based on their approach to specifying the behavior of software systems.

The space of software development paradigms may be envisioned as shown in Figure 3-2, where different paradigms are depicted as regions on a plane of language features. (The features themselves are not shown in the figure.) The vertical placement of a feature in the plane is determined by its level of abstraction: very low-level features, such as those that might be found in a machine or assembly language, are located well below higher-level features, such as might be found in a language like Prolog.

The figure is intended to show that there are a number of software development paradigms available, that they tend to consist of language features at roughly the same level of abstraction, that they may or may not contain features in common with other paradigms, and that the paradigms themselves may have different relative levels of abstraction, i.e., some paradigms are higher-level than others. The shaded area in the figure indicates the focus of my research for this thesis: relatively high-level language features from a number of different paradigms. For view features within this shaded area, my goal is to

[1] It can certainly be argued that FORTRAN is fundamentally different from C and Ada, but the fact remains that, for better or for worse, the same kinds of tools (views) that are applied to C and Ada are typically applied to FORTRAN, too. In that important sense, the three languages are largely similar. (Further clouding the matter is the fact that FORTRAN undergoes a rather drastic metamorphosis every decade or so, so it's not entirely clear what language is meant by the simple name "FORTRAN." As Gene Cordell has remarked, "I don't know what language they'll be using in the year 2000, but I do know it will be called FORTRAN.")

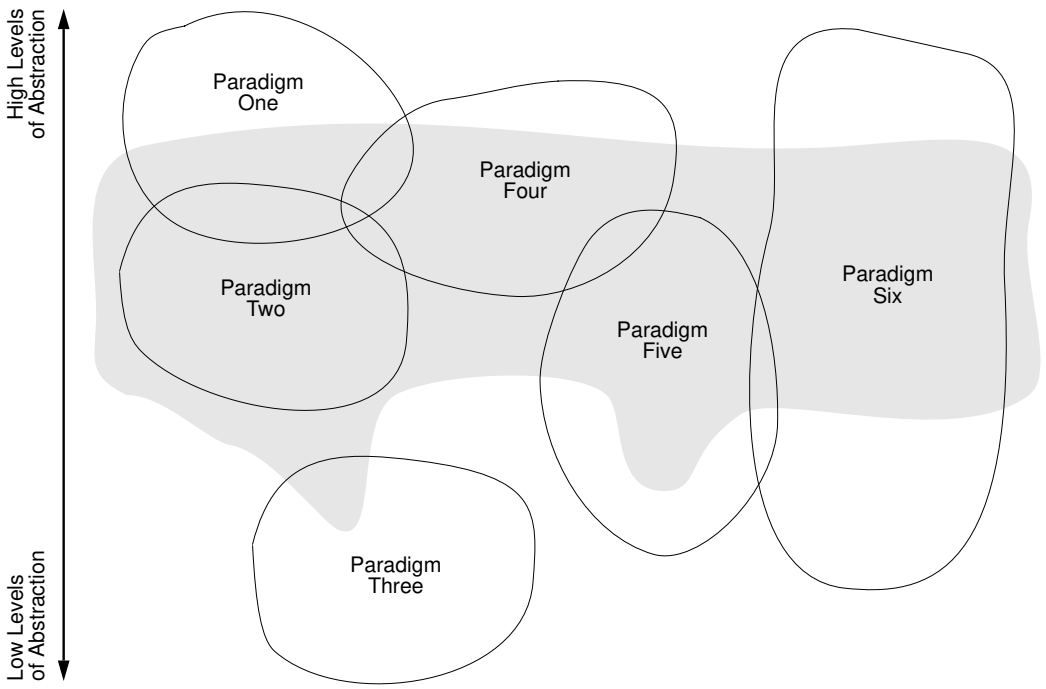


Figure 3-2: Figurative depiction of software development paradigms and their relative scopes and levels of abstraction. The shaded area suggests the breadth of my model of system views.

develop a CR-based MVDE with behavior like that I described in Chapter 1 (Section 1.4).[2]

Although the shaded area tends to encompass features at an "above-average" level of abstraction, it makes a couple of noticeable forays into areas that are comparatively low-level. It is an unfortunate fact of life that some low-level features are present in many common "'high-level" views, and to exclude these popular (but primitive) features would limit the expressiveness and the invertibility of view-CR mappings. Two examples of such low-level features are `goto`s (for control flow programming paradigms) and semaphores (for parallel programming paradigms).

Figure 3-2 is an informal depiction of the kinds of language features I am interested in supporting, i.e., of my model for views. In the next section, I make this intuitive description more precise.

3.3 A Model for Views of Software Systems

My model for views is based on three general paradigms:

- The first is the **sequential control flow** paradigm, which is the most commonly used paradigm in computing today. It encompasses programming language views

[2]This figure should not be taken too literally. For example, there is no reason why paradigms must be convex or why they cannot be nested. The figure is intended only to indicate the existence of different levels of abstraction, of a variety of features in programming languages, of various paradigms of programming, and of how my model of views relates to these things.

such as FORTRAN, Algol, Lisp, BASIC, Pascal, C, etc. It also encompasses the higher-level views often derived from these languages, including call graphs, functional decompositions, program profiles,[3] and test coverage measures.

- The second paradigm is that of **dataflow-based computation**, which is of increasing interest as a method for achieving high-performance fine-grain parallel computing. Views within this paradigm frequently have concrete syntaxes that are similar to those of sequential control flow views, but the underlying semantics of the views are quite different. In particular, parallel execution of subexpressions is limited only by data dependencies, and subprogram calls are non-strict. Examples of this approach to computation include the languages Id [9], Haskell [83], and Lucid [10, 215]. Dataflow is also the basis of a number of high-level design methodologies, e.g., Structured Analysis [177, 178].

- The final paradigm is **parallel control flow** computing, which offers software developers explicit control over the creation of multiple processes and the communication between those processes as they run. Views that correspond to this paradigm frequently contain all of the features of sequential control flow views, plus additional features for the expression of parallelism and interprocess communication.[4] Such views include the programming languages CSP [87], Ada, Concurrent Pascal [27], and Occam [20], as well as the more abstract languages of Statecharts [79], nondeterministic FSAs, and Petri nets.

I selected these paradigms as the basis for my view model for two reasons. First, they collectively encompass a great number of views that are currently in widespread use. Such views include conventional programming languages, call graphs, high-level dataflow diagrams, and Petri nets. Second, they contain the necessary features to support new classes of languages (i.e., views) that are not yet widely employed, but that may well form the foundation for common views in the future. Among such features are explicit parallelism, data-driven computation, and non-strict subprogram invocation. These features are likely to assume greater importance as the use of multiprocessors becomes increasingly common.

The following sections identify the characteristic features of these paradigms that together form a constraint on the minimal expressiveness of a CR for an MVDE.

3.3.1 The Sequential Control Flow Paradigm

Languages (views) based on sequential control flow depend on the following semantic features:[5]

[3] This refers to profiles based on how frequently a statement is executed or how many times a subprogram is invoked. Lower-level profiles, such as those based on how much CPU time is spent in particular statements, fall outside the bounds of the model because of their low level of abstraction.

[4] One might be tempted to say that my model is based on only two different computational paradigms, because the parallel control flow paradigm subsumes the sequential control flow paradigm. It is convenient to examine these paradigms separately, however, because a sequential approach to a problem is frequently quite different than a parallel approach to the same problem.

[5] Each feature listed in this section has a corresponding tag, which I use when I refer back to these features. The tags are all of the form Sn, where S is a short string indicating which language type gives rise to the requirement for the semantic feature, and n is the number of the requirement within that language type. Hence, **SCF1** is the first requirement from the group of sequential control flow languages.

SCF1 Sequential, deterministic control flow constructs: sequential statement ordering, conditional branching, iterative loops, gotos, and subprogram calls. Conditional branches may be two-way (if-then, if-then-else) or multiway (e.g. Pascal's case statement, C's switch statement). Iterative loops have a single entry point, an arbitrary number of return-to-top or exit points (e.g., C's break and continue statements), and a single loop-termination test that may be located at the top of the loop, at the bottom of the loop, or anywhere within the loop. Gotos are restricted only in that they are not allowed to transfer control across subprogram or task boundaries. Subprograms are called synchronously, return zero or more values, and may employ recursion.

SCF2 Identifiers and scopes. Identifier names are statically scoped, may be declared explicitly or implicitly, and need not be unique. Identifiers used as variables refer to values in memory; the details of the values and the memory are explicitly undefined by the model — see Section 3.4. Subprograms may declare local variables; these exist only for the duration of a particular subprogram invocation. Scopes may be disjoint or may overlap. If two scopes overlap, it need not be the case that one is contained inside the other.

SCF3 Operators: assignment, arithmetic, and relational; and **expressions** employing variables, these operators, and parentheses (to allow users to specify the order of evaluation of subexpressions). Multiple simultaneous assignments are allowed (as in CSP[6]).

SCF4 Files. There are no semantics associated with a file in and of itself, but the model does contain the restriction that the intersection of the contents of any collection of distinct files must be empty. That is, no component of a system is contained inside more than one file.

SCF5 Relationships between system components. The model explicitly recognizes that there may be important relationships between components in the model, relationships that are otherwise inexpressible. For example, there may be a compilation dependency relationship between a C source file and the header files it includes. Such relationships are directly expressible in the model, but there are no semantics associated with such relationships. Relationships may be n-ary.

SCF6 Unexecutable ancillary information, which encompasses program comments, external documentation, project management information, etc. This information can be characterized as being "about" the executable part of a software system, but not really part "of" it. It differs from the features above in that its removal from a software system cannot affect the execution of that system. For the purposes of the model, this ancillary information is simply arbitrary text. This restriction results in no loss of generality, because arbitrary information (e.g., images, sounds, etc.) can be encoded textually. The model assigns no semantics to this auxiliary information.

[6]Throughout this thesis, any mention of CSP refers to that language described by Hoare in his 1978 *CACM* paper [87], not to the language described by his 1985 book of the same name [88].

3.3.2 The Dataflow Paradigm

Dataflow languages include Lucid, Id, and Haskell,[7] and although dataflow languages offer a different approach to computation than do control flow languages — typically a more declarative approach — they still tend to offer software developers a set of semantic concepts that is at roughly the same level of abstraction as those offered by control flow languages. This conceptual kinship reinforces the notion that an MVDE should support both types of views.

The important semantic features offered by these languages (in addition to those they share with control-flow languages) are the following:

D1 Data-driven execution. This is the hallmark of a dataflow language, which allows computations to proceed in any order consistent with the (dynamic) availability of the necessary operands. An inherent consequence of data-driven execution is implicit parallelism, because two or more operations are allowed to proceed in parallel if the operands for each operation are simultaneously available. Separate flows of control are created anonymously, nondeterministically, and opportunistically and are beyond the control of the dataflow language programmer. As such, there is neither need nor provision for synchronization of parallel execution beyond that afforded by the requirement that an operation's operands be available before that operation may proceed. In short, the only parallelism supported is fork-join parallelism.

D2 Non-strict subprogram calls. A call to a subprogram S with parameter p is *strict* if $S(p) = \perp$ whenever $p = \perp$. If it is possible that $S(p) \neq \perp$ when $p = \perp$, the call to S is non-strict. In particular, if evaluation of S may terminate even if evaluation of p does not, the call to S is non-strict.[8] Dataflow computation typically

[7] Haskell is viewed by its designers as a pure functional language, but its non-strict semantics, its allowance for implicit parallelism, and its absence of support for side-effects puts it in the same league as dataflow languages. In fact, because of these similarities, research on dataflow languages and functional languages frequently overlaps (see, for example, the motivation behind Lucid [215, p. ix–xii] and the collection of papers edited by Thakkar [211]).

[8] Procedural languages typically use strict calls, while dataflow languages usually employ non-strict calls. For example, subprogram calls in C are strict, so this program will never terminate:

```
int f(int x) { return 1; }    /* always returns 1 */
int g(void)  { return g(); }  /* infinite recursion */
main() { return f(g()); }     /* never terminates */
```

In this program, there is really no *need* to evaluate `f`'s argument — `f` never even looks at it — but because C has strict subprogram calls, evaluation of `f` cannot proceed until evaluation of `f`'s argument is complete. Thus, `main` will never terminate.

In a non-strict language like Lucid, however, this seemingly equivalent program behaves quite differently:

```
main where
  f(x) = 1;     // always returns 1
  g = g;        // infinite recursion (conceptually)
  main = f(g);  // returns 1 immediately
end;
```

Conceptually, the definition of `g` in this program, too, exhibits infinite recursion, but because subprogram calls in Lucid are non-strict, this program produces the value 1 immediately. (Technically, Lucid programs map infinite streams of input data into infinite streams of output data, so this program really produces not a single 1, but an infinite stream of 1s.)

employs non-strict subprogram calls.

3.3.3 The Parallel Control Flow Paradigm

Languages like CSP, Ada, and Petri nets allow programmers to work with concurrency explicitly. Given that the model for software systems must provide support for implicit parallelism, it is only reasonable that it also provide support for explicit parallelism. The need for such support motivates the addition of the following features to the model:

PCF1 Dynamic process creation. Many parallel programming systems (e.g., Ada, the Unix shells sh and csh) begin execution with a single sequential process that may spawn new sequential processes that may themselves spawn new processes. Other systems (e.g., CSP) may execute multiple processes from the outset. The set of processes that may be created is statically limited in some views (e.g., CSP, Petri nets, Statecharts), while in other views an unlimited number of processes may be created at runtime (e.g., in Ada). My model for views supports the most general of these possibilities: any number of independent processes may start up when the software system is invoked, and each process may spawn new processes without limit.

PCF2 Interprocess communication. The model allows for interprocess communication (IPC) for the purposes of mutual exclusion, synchronization, and client-server interactions (e.g., remote procedure calls).[9] The model contains "generalized tasks," which are based on Ada tasks. A generalized task is a process (possibly created dynamically) with a well-defined set of messages that it can receive (entry points) and, optionally, data that only it can access. Only generalized tasks can receive messages.

The model supports rendezvous: bidirectional message passing where the message sender blocks until the message recipient replies; if no reply is expected, the sender blocks only until the recipient has received the sent message. Messages may be of unlimited length. The sender of a message must know the name of the receiver, but the message recipient need not know the name of the sender. Process names may be passed back and forth as part of a message.

All messages for a task are automatically queued for the recipient in buffers of unlimited size; there is a single queue for each task. When a task is ready to receive a particular message (i.e., an entry call), it receives the next instance of that particular message that is waiting in the queue. If a task expects to receive a message and none is available, it blocks until one is available.

PCF3 Nondeterministic execution. Nondeterministic execution arises naturally in parallel systems, particularly when a task can service more than one entry call, but its use in views is actually more general than this: consider Petri nets (which support only fork-join parallelism) and nondeterministic FSAs (which typically contain only a single flow of execution at a time), both of which offer nondeterministic branching.

[9] It is also expressive enough to implement coroutines.

Tag	Feature
SCF1	Sequential, deterministic, control flow constructs
SCF2	Identifiers and scopes
SCF3	Operators and expressions
SCF4	Files
SCF5	Relationships between system components
SCF6	Unexecutable ancillary information
D1	Data-driven execution
D2	Non-strict subprogram calls
PCF1	Dynamic process creation
PCF2	Interprocess communication
PCF3	Nondeterminism in execution

Table 3-1: Summary of semantic features in the model for software systems.

My view model supports guarded nondeterminism in the style of CSP and Ada, whereby nondeterminism can be used to determine which of a set of competing statements will be executed at a particular point in a program. Each competing statement may be controlled by a guard (a boolean expression), and at runtime when a nondeterministic choice must be made, only those statements with a true guard are eligible for execution. My model is more general than the CSP and Ada model, however, because the model allows any number of statements with true guards to be executed. Hence a nondeterministic construct can be used to initiate fork-join parallelism.

3.3.4 Summary

Table 3-1 summarizes the basic features of the view model for software systems. This model allows for the existence of different computational paradigms, and it includes both control- and data-driven execution, both serial and parallel execution, both deterministic and nondeterministic execution, both synchronous and asynchronous subprogram calls, both strict and non-strict subprogram calls, as well as traditional control flow constructs. It also recognizes the existence of unexecutable information, such as arbitrary n-ary relationships between system components and ancillary (textual) information.

3.4 The View Model and Data Structures

This model of software systems is in many ways substantially more expressive than are the implicit models employed by current development environments. Nonetheless, there are important aspects of software systems that are not included in this model.

The most important such aspect is support for data structures. In fact, beyond the explicit recognition of the existence of data values and variables that can hold them, this model includes no mention of data at all. Within the model, data values are simple integers, and variables are either global or local to a subprogram. Pointers, static variables, and dynamically allocated data (other than local variables for subprogram invocations) do not exist.

It may seem counterintuitive, but support for data structures in a CR is largely orthogonal to support for program structures and executable semantics. That is, issues of subprograms, scopes, determinism/nondeterminism, strictness/non-strictness, etc. — the issues that this model *does* address — are largely independent of the data structures that are manipulated. Similarly, much work has been done on translating between different views of data structures without much regard for how those structures are manipulated by the programs in which they find themselves [85, 69, 102, 126, 217, 201, 31, 222]. Thus, a model of how a program is structured and how it executes can be largely divorced from the data structures it manipulates. Modeling data structures is certainly important, and support for data structures could be added to the model I propose here, but that is an aspect of views I have chosen not to address in this thesis. This is analogous to research on systems for algorithm animation [29, 193]. Such systems focus entirely on what happens to the *data* in an application, completely ignoring the control structures within the application that are responsible for generating and manipulating the data.

A consequence of this decision is that the model is silent with respect to object-oriented languages like Smalltalk and C++. It makes little sense to discuss support for such essential object-oriented features as dynamic binding if there is no way to distinguish data types in the model. I revisit this topic in Chapter 8, where I discuss possible extensions to this model of views.

3.5 Implementing the View Model

I noted at the outset of this chapter that the view model I developed provides a constraint on the minimal expressiveness of a CR for an MVDE. The relationship between a CR and the model is more direct than that, however: a CR is an *implementation* of the model.

Conformance with the view model is but one of a number of requirements that a CR must satisfy, however. Another is that it be free of biases toward particular kinds of concrete syntax, because views that fit the model range from graphical views like Petri nets to textual views like Ada. A third is that it be executable, because the MVDEs in which I am interested must support dynamic views of a software system. (As I noted in Chapter 1, most dynamic views are based on existing static views, so satisfying the requirement for executability is tantamount to satisfying the need to support dynamic views.)

A final constraint on the design of a CR is that the CR must facilitate the development of bidirectional mappings between the CR and the views that use it. This requires that a good representation have the right set of semantic primitives, a set that is well matched to the abstraction level(s) of the class of views that it is designed to support (cf. Section 3.2). Primitives that do not correspond to this set of semantic notions may be safely eliminated, and in fact doing so considerably simplifies both the representation and the mappings from it. For example, the designer of a representation need not provide support for unusually high level languages like Prolog or SETL [183], nor for unusually low level programming languages like assembler.

This does not imply that a representation need not cater to high level views at all. Far from it. High level views that are *abstractions* of languages like FORTRAN, C, and Ada — for example, Petri nets, call graphs, and functional decompositions — are perfectly reasonable and are entirely worthy of support. This is far different from providing support

for Prolog, however, which would require that a representation understand unusually high level *semantic primitives* like unification and depth-first search.

A good representation must be able to directly and accurately reflect "what is going on" in a software system, and it must do this in a way that is relatively easy for views to map into *and* map out of. The forward-mapping problem is simple and has been solved for decades. Virtually all computer systems allow the creation of an executable image that is composed of binaries generated from different source languages: the different source languages are the views, the different compilers are the mapping functions, and the machine language is the common representation. This same organization, albeit usually at a higher level of abstraction, is also the one on which multiparadigm languages are founded.

The back-mapping problem is considerably more difficult, and making back-mappings tractable is the primary challenge in the design of a suitable CR. To facilitate mapping from the representation to views, the abstraction level of the semantic primitives must be roughly the same in both. This match between levels of abstraction is the reason why using a low-level representation such as a Turing Machine or assembly language is unsatisfactory — the back-mapping problem becomes too difficult.

3.5.1 Choosing Semantic Primitives

How, then, does one choose the appropriate set of semantic primitives to put into a CR? One possible approach is to examine a large set of views from the class one wishes to support, then simply have the CR embody the *union* of the semantic primitives in that set of views. The flaw in this union-of-features strategy is that back-mapping becomes complicated by the need to distinguish between a possibly large set of semantically similar constructs. For example, iterative loops typically come in a variety of flavors (test-at-the-top, test-at-the-bottom, test-in-the-middle), but in essence there is only one concept that needs to be expressed, that of an iterative loop. Explicitly representing each minor variation on the theme makes the CR unnecessarily difficult to understand and obscures the essence of "what is going on."

The antithesis of the union-of-features approach is to put into the CR only those semantic features that fall in the *intersection* of the set of features exhibited by the set of views examined. This approach avoids the proliferation of similar semantic primitives, but it also prevents writers of view-to-representation mappings from expressing critical distinctions between fundamentally dissimilar concepts. For example, it is difficult to determine what form of subprogram call should be included in an intersection-of-features representation that must support views with strict calls (only) as well as views with non-strict calls (only).

My approach is a middle ground between these two extremes. Regrettably, this implies that there is no simple test to determine whether a particular semantic feature should be incorporated unchanged into a CR or whether it should be conceptually merged with other similar features and expressed in some "neutral" fashion. Retaining the feature in its original form simplifies mappings from views to the representation, but runs the risk of complicating back-mapping in the same way as does the union-of-features approach. Merging the feature with similar constructs simplifies the representation, but it may complicate back-mappings in the same way that the intersection-of-features approach does.

The decision as to which semantic features should be in the canonical representation

and which should not must therefore be made on a case-by-case basis. If a view feature is at the appropriate level of abstraction and is not more or less directly expressible in the CR, and if it would be difficult to recognize the feature in the CR in some approximate form, then it is a good candidate for inclusion in the CR (though possibly in a modified form so that other, similar, features can be merged with it). On the other hand, if a view feature can be mapped into an existing CR structure (or recognizable set of structures) in a more or less straightforward fashion, and if back-mapping from this CR structure or set of structures to the original feature is not too difficult, then there is no need to augment the CR, and the view feature should not be added to the CR.

In the ensuing sections, I discuss four specific representation problems, and I sketch how each is handled in SPGs. An examination of these particular problems is enlightening, because the issues they address — looping constructs, subprogram calls, scopes and name bindings, and interprocess communication — are conceptual mainstays of programming; no CR that fails to resolve these problems can possibly be acceptable as the basis for an MVDE. Furthermore, the solutions to these problems are interesting in their own right, demonstrating as they do one case in which a set of language features should be merged, a second case in which they should *not* be merged, a case in which a language feature should be omitted from the CR entirely, and a case in which a special provision should be made for a language feature that is conceptually primitive, yet is too prevalent to ignore.

In the material that follows, I offer only outlines of how I approach these representation matters using SPGs. A detailed solution to each problem is featured in Chapter 5, Sections 5.2.1–5.2.4.

3.5.2 Representing Iterative Loops

Consider the three iterative looping constructs in C: for loops, while loops, and do loops. Rather than have a separate structure in a CR to represent each of these kinds of loop, it is better to generalize the notion of an iterative loop to that of a looping construct with an exit test. This more general construct can then be added to the CR, and each form of loop in C can be mapped into this general construct directly. For back-mapping, the writer of the C view would have to examine the structure of the general loop in the CR to see where the exit test was located, and would then have to determine how to best express that in C using one of the available choices. This is not an onerous burden for the writer of the C view, and it has the critical advantage that *other views will also be able to recognize the loop and its semantics*, even if they have no notion of for, while, or do loops at all! Fundamentally, "what is going on" is a loop, and that is what the CR must accurately represent; the particular variant of loop is of little concern.

Some languages, early versions of BASIC and FORTRAN among them, have only one kind of built-in loop, the for loop. Yet the concepts of a test-at-the-top loop and a test-at-the-bottom loop still exist in these languages; they must be implemented using gotos. Nonetheless, programmers in these languages have no difficulty in recognizing these loop variants when they see them in source code. There is no difficulty in mapping a goto implementation of a test-at-the-top or test-at-the-bottom loop to a generic iterative looping construct in a CR, and once in the CR, "what is going on" becomes immediately apparent to other views that, perhaps, have no notion of a goto. In this sense a carefully chosen set of semantic primitives in the CR can function as effective intermediaries between views that

seem to differ radically, e.g., between views with loops but no gotos and views with gotos but no loops.

3.5.3 Representing Subprograms and Calls

Representing subprograms is rendered fairly straightforward by the fact that there is wide agreement about their structure: a subprogram is an instantiable code segment customizable with zero or more data values (parameters) that computes and returns to the caller zero or more values. What complicates matters somewhat is the way in which actual parameters are passed in and results are returned.

Because the view model deliberately avoids discussion of most of the issues surrounding data structures, it suffices for a representation to choose an arbitrary technique for passing values in to subprograms and returning results from them. That is, rather than struggling with the problems of trying to unify by-value and by-reference parameter passing schemes,[10] a representation can simply choose to pass everything by value;[11] refinements on this approach can be safely deferred until the view model is expanded to deal with data in a more comprehensive manner. Note, however, that the view model does provide support for the passing of subprograms themselves as parameters, so this must be allowed by the representation. Fortunately, this can be easily achieved by assigning to each subprogram a unique identifier; the identifier can then be passed (by value) as data to and from subprograms. This is equivalent to the approach adopted in C, where functions *per se* may not be passed to and from other functions, but the *addresses* of functions may be.

Passing arguments and results by value puts to rest one semantic point of contention for subprograms, but it fails to address an issue more central to the view model: that of strictness. In Section 3.3.2 I pointed out that languages centered around control flow are generally strict with respect to their subprogram calls, while dataflow languages are generally non-strict. There is no way to unify these fundamentally different concepts — before evaluating the body of a subprogram you either evaluate its actual parameters or you don't. A good CR must therefore support both kinds of calls. However, it is possible to generalize the notion of strictness by allowing it to be specified on a per-parameter basis, rather than a per-call basis. By doing so, a CR can offer support for views in which calls to subprograms may contain a mixture of strictly and non-strictly passed parameters. Provided that the CR clearly distinguishes between parameters passed strictly and those passed non-strictly, this generalization increases the expressiveness of the representation without decreasing the ability of a view to determine "what is going on."

3.5.4 Representing Scopes and Name Bindings

A fundamental concept in programming languages is that of scopes and their use for binding names to values. Program representations that lack support for scopes suffer from important

[10] The omission of by-name parameter passing is not incidental. Although such parameters can be represented as thunks in the usual manner [159] and a special annotation can be associated with them so that views can easily identify the role they play, in the general case — in particular, in conjunction with recursion in the calling routine — thunks act like continuations, and continuations are not part of my view model.

[11] This choice is motivated by the fact that by-value is the fundamental parameter passing technique employed by dataflow programming languages. For control-based languages, there is no "obvious" choice, because by-value and by-reference are in some sense equally natural.

drawbacks, as noted by researchers on the Ergo project:

> `Term`s (one may think of them as abstract syntax trees) have a crucial weakness, namely that they do not carry information about the name-binding rules of the object-language. This makes it impossible to write, for example, *one* function for substitution that is syntactically correct for *all* object-languages. ... From this and other examples it is clear that one would like to include information about the binding properties of object-language constructs in the representation of programs. [113, p. 28].

Scopes themselves present no particular difficulties. A scope is just a region of a program and may be directly represented as such. The matter of name-binding, however, is substantially thornier.

In my early work on SPGs, I proposed that each scope have associated with it a set of name-binding rules that collectively defined how names were bound in that scope [135].[12] This seemed like a natural and flexible approach to an ubiquitous concept, but it turned out to be untenable. There were two reasons for this:

- **Name-binding rules vary too much across views.** Some views scope names lexically, some do not. Some views allow overloading of names, some do not. Some views allow names to be implicitly imported and exported across scope boundaries, others require that such movement be explicit. Coming up with a simple set of rules for such a wide variety of conventions would have been difficult, and providing enough flexibility to allow for the creation of new scoping rules to satisfy as-yet-unimagined views would have been harder still. The only solution seemed to be creation of a new "name-binding behavior specification language," but the existence of such a language within a CR would have made mapping from the CR to views substantially more difficult. Practically speaking, it would have made it *too* difficult. (An example of the complexity of such a language can be found in Reiss' work on the automatic generation of symbol table routines for use in compilers [162].)

- **Defining the name-binding rules for a scope created in one view and then modified in another view was problematic.** Suppose a scope S is created in view V_1. Naturally, it assumes the name-binding conventions of V_1. Now suppose that S is edited using some other view V_2. What name-binding behavior should S now exhibit? Should it continue to bind names as in V_1, should it henceforth behave like a scope in V_2, or should it exhibit some combined behavior that reflects its multiple-view editing heritage? Regardless of the solution chosen, it is difficult to justify it as the single "correct" solution.

Based on these considerations, I came to the conclusion that a CR should deal only with unambiguous names. In effect, all identifiers within a CR must be unique, but provision must be made for views to easily map the unique names employed in the CR back to the programmer-defined names in each view. The result is that name-binding within the CR is trivial, and views are responsible for mapping between view-specific name binding rules and the unambiguous references used in the CR.

[12] These rules were more specific than those defined by Pratt for identifier associations [159, p. 185], but they were similar in spirit.

It may seem unnatural to place the responsibility for name-binding entirely on the views in an MVDE, but in fact this separation of name-binding concerns between the CR and the views is precisely the same as that between environments and stores in the denotational semantics of programming languages [208, 154]. That is, in terms of denotational semantics, the CR handles stores (values in memory), while views handle environments (name-binding).

In fact, the Ergo project adopted an equivalent approach by deciding to rely on the simply typed λ-calculus as a representation language. In the λ-calculus, there is only a single name-binding rule (abstraction), and view implementors are responsible for mapping between view-specific semantics and that of the λ-calculus.

For any particular view, mapping view names into CR names is not difficult, because the view can always identify the point of declaration of any name used in the view (even if the declaration is implicit). Therefore, the use of globally unique names inside a CR does not complicate the mapping from views to the CR. The back-mapping problem, however, is not so simple. The representation can be designed such that it is trivial to map from a use of a CR identifier to the CR declaration site for that identifier, but this does not solve the problem of mapping from the use of a CR identifier to its view-specific name in that context. The reason for this is that the view may not have a scoping rule that corresponds to the mapping from the use site to the declaration site. This would be the situation if, for example, a view with lexical scoping attempted to display a program that was created using a view where scopes need not adhere to the lexical structure of the program.

In general, mapping from a CR to a view will require that the view try to find a view-specific scoping rule that can account for the relationship between the observed use and declaration. The view must determine which rule was used, because it may affect the presentation of the view. For example, consider this (silly) C++ program:

```
int one = 1;
main()
{
  int one = 1;
  int two = one + ::one;
}
```

In the CR for this program, there are three identifiers, two of which are named "`one`" in the view. It is not sufficient for the view to map back from the uses of the (CR) identifiers to their view-specific names, because both names are the same, "`one`." Instead, the presentation must show them differently (as "`one`" and "`::one`") in the definition of `two`. To do this, it must know that different scope rules were applied to resolve the names: local scope for the first occurrence, global scope for the second one. In general, a view is faced with a particular scoping relationship between a use and a declaration, and it must determine which — if any — of its applicable scoping rules should be used to explain that relationship.

Although the CR proper cannot assist in this task, is is certainly possible to build a library of routines that understand commonly used scoping rules (e.g., local scope, lexically nested scope, file scope, global scope, etc.) and to make this library of routines available to writers of views. Views that use common name-binding conventions (e.g., most languages of the Algol family) can employ the library routines to resolve scoping questions, but views using novel scoping models are free to write view-specific routines that behave as they desire. This approach is as powerful and flexible as the design I originally proposed (the

one based on a set of rules for scoping behavior built in to a CR), but is substantially more flexible, because it can be easily extended by views. Furthermore, this design requires that views know only about their own scoping models, whereas the original proposal required that views be able to cope with any scoping model defined using the rules available in the CR.

3.5.5 Representing Interprocess Communication

Parallel programming languages offer a number of different mechanisms for achieving interprocess communication. An examination of "what is going on" in such languages, however, reveals that IPC is almost always employed for one of the following purposes:

- Maintenance of **mutual exclusion** to shared resources;
- **Synchronization** of relative rates of progress;
- **Client/server relationships**, whereby client processes request services of server processes and server processes return the results of such service requests to clients. The client/server relationship subsumes mechanisms for remote procedure calls.[13]

Unfortunately, these fundamental uses are often obscured by the details of the precise mechanism available in a given language, which can range from semaphores (in Algol 68 [203]) to monitors (in Concurrent Pascal) to unidirectional message passing (in CSP) to bidirectional message passing (in Ada and Occam). These primitives vary both in their level of abstraction (semaphores are the assembly language of IPC, while Ada's tasks hide all low-level details of interprocess communication) and in their precise semantics (monitors require FIFO message queuing, while semaphores, CSP messages, and Ada tasks specify only that queuing be "fair"). How, then, should IPC be incorporated into a CR?

Setting aside semaphores for moment, there are important similarities behind monitors, CSP messages, and Ada tasks. In each case mutual exclusion is achieved by encapsulating shared resources — typically data — such that only a single sequential process can access it. In each case the runtime environment is responsible for scheduling access requests such that mutual exclusion is maintained. In each case there is a fixed and well-defined set of entry points (monitor procedures, CSP message names, Ada task entries) to the sequential process that can access the shared resources. Support in a CR for IPC can be based on this general view-independent foundation.

Furthermore, the design of a CR can take advantage of the observation that parallel programming languages almost invariably offer only *one* form of IPC. This is in striking contrast to the situation with loops, where it is not at all uncommon for a language to provide three rather similar structured looping forms, in addition to the unstructured forms that can be built using `goto`s. Because most languages provide only one form of IPC, writers of CR-to-view mappings need only be able to detect the general notion that IPC is taking place; from that point forward they know what language structure they must map it into. As a result, the CR need not specify in excruciating detail the form of the IPC, because the view writer is compelled to use whatever is available. For example, the CR need not go

[13] "Remote" in this case merely means "in a different process," not necessarily "on a different processor."

to great lengths to specify the scheduling strategy,[14] because a view is likely to have only one scheduling strategy to map the IPC into. In short, developing a mapping from a CR manifestation of IPC to a view manifestation of IPC is simplified by the dearth of choices that the view writer is likely to have in the syntax and semantics of the view.[15]

Semaphores present a special difficulty, because they are unstructured, low-level concurrency primitives that have little in common with the higher-level constructs (above) that their shortcomings historically motivated. Because they are so primitive, it is tempting to omit them from consideration when designing the IPC mechanisms for a CR, but unfortunately, semaphores continue to be widely used, even in languages of contemporary design [148, 100, 75]. As a result, ignoring semaphores is simply unrealistic — they are just too prevalent to pretend that they don't exist. This is an example of a situation in which the CR must provide explicit support for a language feature that is otherwise too primitive for inclusion simply because the feature is widely used.

Fortunately, it is a simple matter to simulate semaphore behavior using monitors [20, p. 65-66], so semantic support for semaphore behavior calls for no special provision in the CR. To facilitate mapping from the CR to views, however, it is prudent to specifically provide a special tag for emulated semaphores so that view writers can quickly recognize them; this is the approach I take with SPGs.

3.6 Summary of Constraints

The considerations raised in Sections 3.3 and 3.5 dictate the following constraints on a CR for an MVDE:

- The semantic primitives offered by the CR must be at roughly the level of abstraction of current high-level programming languages. They must be expressive enough to represent the semantics of the characteristic features of sequential control flow computing, dataflow programming, and parallel control flow computation.
- These primitives must be carefully designed so that essentially similar view features map to a single CR feature and essentially dissimilar view features map to different CR features.

These are the criteria against which all candidate CRs must be measured. Before expending the effort to develop a novel representation, it is important to examine existing representations to see if they could serve as the CR for an MVDE.

[14] For execution purposes, however, the strategy that it employs must be consistent with the semantics of many different IPC mechanisms. If it is not, then mapping view IPC semantics into CR IPC semantics becomes difficult, and the CR fails to meet the expressiveness criterion.

[15] I imply here that when a view writer maps from a semantic concept in the CR to its manifestation in the view, the writer is likely to *approximate* the semantics if they cannot be expressed exactly. For example, a generalized task in a CR is likely to be mapped into a task in an Ada view, even though the scheduling strategies have slightly different semantics. Semantics approximation is not the only recourse, however, and I discuss this issue further — along with its alternatives — in Chapter 6.

3.7 An Evaluation of Existing Canonical Representations

During their design of the Plan Calculus [170], Rich and Waters identified "four key properties essential to a knowledge representation" for software systems. These are "canonical form, language independence, convenience of manipulation, and expressiveness" [173, p. 23]. Their concerns mirror my own, although their interest in convenience of manipulation, which is founded on their desire to perform sophisticated program analyses, is focused differently than mine, which concentrates on the representation's suitability as both source and target of mappings to and from views. In this chapter I considered each of these factors, except that of a canonical form. I address that topic in Chapter 8.[16]

In Chapter 2 (Section 2.4.3) I discussed a plethora of existing canonical representations, dividing them into the general categories of abstract syntax trees, abstract syntax graphs, the Plan Calculus, and a variety of miscellaneous representations. None of these existing representations meets the twin requirements of (1) being expressive enough to support the view model outlined in Section 3.3, and (2) facilitating the development of bidirectional mappings between the CR and views that are consistent with the model.

Abstract syntax trees and graphs, as their names suggest, are inherently biased towards a particular abstract syntax In an MVDE, however, it is to be expected that different views may have wildly different syntaxes, both concrete and abstract (requirement S1). Syntax-based representations, then, are inappropriate for the CR in an MVDE, because they are too limiting in the range of views they adequately support.

The Plan Calculus suffers from no syntactic bias, but its approach to representing "what is going on" is at a higher level of abstraction than is the view model (and hence the paradigms from which that model was derived). For example, its representation of control- and dataflow depicts only the partial ordering necessary to preserve the correct executable semantics, even if a view itself chooses to include unnecessary dependencies (e.g., gratuitous statement sequencing). In this respect the Plan Calculus is similar to Program Dependence Graphs, which abstract away too much information and thus complicate mapping from CRs to views. As another example, iterative loops in the Plan Calculus must be represented as recursive subprogram calls. Although this faithfully preserves the executable semantics, it complicates the process of recognizing "what is going on" from a view writer's perspective. If a programmer writes a loop as an iterative construct, that programmer will be quite surprised if the view chooses to display it as a recursive call! Iteration may be semantically equivalent to tail recursion, but to most programmers, these concepts are distinct, and a good representation must not obscure that distinction.

I described the shortcomings of the miscellaneous canonical representations at the end of Section 2.4.3. Each representation either lacks adequate support for the creation of tractable mappings to and from views (e.g., Program Dependence Graphs and their derivatives), provides insufficient predefined semantics (e.g., typed hypermedia), or fails to allow dynamic views (e.g., Lubar's General Design Representation).

Given that general-purpose multiple-view editing of software systems through user-definable views has never really been attempted, it is not unsurprising that none of the existing CRs is well suited to supporting the view model I developed in this chapter. It

[16] In brief, a CR offers a *canonical form* for a semantic concept if there is only one way to express that concept in the CR. This property is highly desirable, because it immensely simplifies the problem of mapping from the CR to views. Unfortunately, it is also extremely difficult to achieve.

is possible to design a representation to better meet the unique challenges of an MVDE, however, and in the following chapters I describe Semantic Program Graphs and how they provide an improved basis for multiple-view software development.

Part II

Semantic Program Graphs

Chapter 4

Semantic Program Graphs

A Semantic Program Graph (SPG) is a graph structure for representing information about software systems. SPGs are so named to emphasize the fact that they are designed to effectively represent the *semantics* of a software system — to clearly indicate "what is going on." In this way SPGs differentiate themselves from other graph-based program representations, including abstract syntax graphs (which focus, naturally enough, on syntax) and program dependence graphs (which are designed to effectively represent control and data dependencies).

The information embodied by an SPG falls into three general categories:

- **Structural components** of the system, including files, scopes, subprograms, parameters, statements, etc.
- **Executable information**, which specifies how the system being represented behaves when it is executed. Examples of executable information include which assignments to perform, which subprograms to call, and which data values to propagate.
- **Unexecutable information**, primarily documentation.

In designing the executable aspects of SPGs, I was influenced by existing work in graph-based program representations. In particular, I was impressed by the the Combined model of computation advanced by Treleaven and his colleagues in their work on an architecture that supports both control- and data-driven execution [212]. I also took ideas from the Tagged-Token Dataflow Architecture [9], most notably in the area of support for non-strict subprogram calls. My approach to unexecutable information is based on concepts underlying hypermedia systems [43], and my use of hypergraphs instead of conventional graphs is a direct reflection of how my thinking was shaped by ideas from the hypermedia community.

In this chapter, I provide a comprehensive description of the syntax and semantics of SPGs, and I provide background information on the design decisions I made during their development. The resulting narrative is quite long, and it contains a large portion of very technical material, much of it of the kind generally found only in language reference manuals. Readers uninterested in the details of SPGs may wish to read section 4.1 only, which provides a high-level overview of the primary features offered by SPGs.

Lest the overall structure of the thesis become obscured by the many details in the sections that follow, it is worth reviewing how the material in this chapter relates to that in

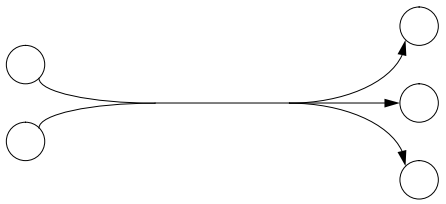


Figure 4-1: A hyperedge with two source nodes and three destination nodes.

Chapters 3, 5, and 7. Chapter 3 defined a model for views and laid down a set of criteria that a CR for an MVDE must satisfy if it is to be an acceptable implementation of the model. This chapter defines SPGs. Chapter 5 ties the two previous chapters together by providing an *analytical* treatment of how SPGs satisfy the expressiveness criterion introduced in Chapter 3. Chapter 7, on the other hand, provides a preliminary *empirical* assessment of how well SPGS meet the criteria described in Chapter 3.

4.1 Overview

The purpose of this section is to provide a high-level overview of the components that make up an SPG, to outline how these components are used to represent the static structure of software systems, and to explain how an SPG can be used to simulate the execution of the software system the SPG represents. Being an overview, this section describes some aspects of SPGs in terms that are incomplete and or imprecise. The sections that follow this overview provide a complete, precise, formal definition of the structure and semantics of an SPG.

An SPG is a form of directed hypergraph. A hypergraph differs from a conventional graph in that any edge in a hypergraph may have multiple source nodes and/or multiple destination nodes; see Figure 4-1. Such edges are correctly termed *hyperedges*, but in this thesis I am concerned primarily with hypergraphs, so I usually just refer to them as *edges*. In cases where I need to distinguish between conventional single-source single-sink edges and hyperedges, I refer to the former as *simple edges* and the latter as hyperedges.

Edges are useful for modeling the many *directed* n-ary relationships that naturally arise in software systems. For example, a conditional construct (if-then-else) can be directly represented as an edge with one source and two destinations: the source corresponds to the statement preceding the conditional, and each destination represents the next statement to be executed, depending on which branch of the conditional is taken. Similarly, the relationship between a use site and the definitions that might reach that use can be represented as an edge with each definition site as a source and the use site as its destination.

Each SPG edge contains *branches*, and it is the branches that actually connect the *edge body* to its sources and destinations. A branch running from a source to an edge body is a *source branch*, while a branch running from an edge body to a destination is a *destination branch*. The edge in Figure 4-1 contains two source branches and three destination branches.

Branches connect to nodes at *ports*. There are two kinds of ports, *input ports* and *output ports*. An edge in an SPG connects to each of its sources at an output port (because tokens flow *out* of the source onto the edge) and connects to each of its destinations at an input port (where tokens flow *into* a destination). An input port may be the terminus of any

Component	Represents
Node	Locus of computation; unexecutable information
Edge	Directed n-ary relationship
Branch	Single input or output part of an edge
Port	Connection between a branch and another SPG component
Grouping	Undirected n-ary relationship
Annotation	Unary relationship
Token	Dynamic flow of computation

Table 4-1: Components of an SPG.

number of destination branches, but only a single source branch may originate at an output port. This restriction is not as limiting as it may initially seem, because a single source branch may lead to an edge with any number of destination branches.

Nodes in an SPG are where computation takes place. They are used to represent, among other things, expressions, statements, and subprogram calls. In this sense they are similar to the nodes in traditional flow graphs. They may also represent unexecutable information of virtually any kind. In this sense they are similar to nodes in untyped hypermedia systems.

In addition to nodes and edges, SPGs contain two components not found in most graph structures: groupings and annotations. Groupings are arbitrary sets of other SPG components. They are used to model *undirected* n-ary relationships between elements in an SPG. For example, they can be used to represent regions of programs, and as such they can be used to model concepts such as subprograms, scopes, and files. They can also be used to represent less traditional relationships, such as all the functions written by programmer X, or the set of nodes left uncovered by a particular test suite. Groupings may be nested, so they naturally lend themselves to the modeling of nested scopes, hierarchical file systems, and similar structures.

Annotations are used to express *unary* relationships. For example, a grouping representing a subprogram can be annotated with the name of the subprogram, and an edge can be annotated with its role in a looping construct. Any SPG component (including annotations) can be annotated.

During execution of the software system represented by an SPG, tokens flow through the graph. There are two kinds of tokens: control tokens and data tokens. Control tokens are used to represent control-driven execution, while data tokens represent data-driven execution. This distinction between control and data is also reflected in the edges and ports of an SPG: there are disjoint sets of control and data edges, and disjoint sets of control and data ports. SPGs enforce "strong typing" with respect to control and data: control tokens may only flow through control ports and along control edges, and data tokens may only flow through data ports and along data edges.

These are all the components of an SPG. Table 4-1 lists these components and summarizes what each represents. In the remainder of this chapter, I discuss these components formally and in detail, but it is useful to examine a simple example now to see how the different parts of an SPG work together to represent a program and to simulate its execution.

Figure 4-2 shows a Pascal program that reads in a series of numbers and then prints out their average. Figure 4-3 shows a "pseudo-SPG" that represents this program. The pseudo-SPG bears the same relationship to a real SPG that pseudocode bears to a real

```
{ Program to print out the average of a set of input numbers. }
{ Input terminates when the sentinel -1 is seen. }

program average(input, output);

  var sum: integer;
      n:   integer;
      x:   integer;

  begin
  n := 0;
  sum := 0;

  read(x);
  while x <> -1 do
    begin
    n := n + 1;
    sum := sum + x;
    read(x);
    end;

  write('Average is ', sum / n);
  end.
```

Figure 4-2: Pascal program to compute the average of a set of numbers.

program. It is fundamentally accurate, but some details have been omitted.

It is apparent from Figure 4-3 that an SPG looks much like a conventional flow graph. There are, however, some important differences. One difference is the use of hyperedges in two places in the graph. One hyperedge is used to represent the two possible paths into the loop, one path from above the loop, the other from the end of the loop itself. The other hyperedge is used to represent the test at the top of the loop, one destination branch entering the loop (if $x \neq -1$), the other leading to the write statement following the loop. A second difference between an SPG and a conventional flow graph is that the condition for loop termination (whether $x \neq -1$) is *part of the branch to be taken* if the condition is true. This is in contrast to flow graphs, which would typically contain a node for the purpose of computing the relationship between x and -1.

Each destination branch in an SPG may contain a condition that controls whether tokens are allowed to flow along that branch. Figure 4-3 shows two such conditions. One corresponds to the explicit condition $x \neq -1$, the other corresponds to the implicit condition that is true whenever the explicit condition is false. That is, the Pascal program explicitly contains the test of whether $x \neq -1$, but it also implicitly contains a test of whether it is *not* true that $x \neq -1$. This is not the same as checking to see if $x = -1$, because the test performed by the implicit condition is dependent on the explicit condition. If the explicit condition (i.e., the loop termination condition) is altered, the test performed by the implicit condition must also change, but the implicit condition itself is *unchanged*: it is still true iff the explicit condition is false. Within an SPG, this kind of implicit condition is made

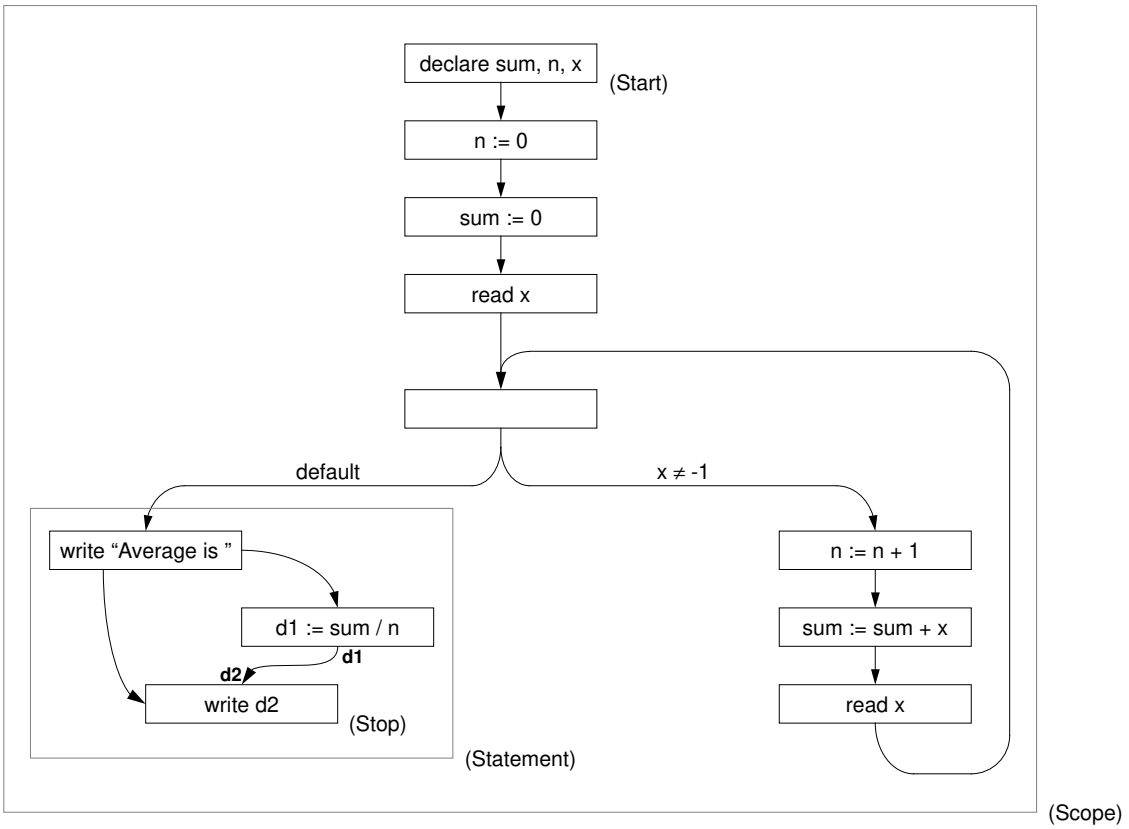


Figure 4-3: Pseudo-SPG for the averaging program of Figure 4-2.

explicit by use of the special condition default, which is defined to be true iff the conditions controlling the other destination branches of the edge are false.

This brief discussion of the topology of an SPG suffices to bring out some of the features that make an SPG uniquely suited for the faithful representation of "what is going on" in a software system. In the program average, control can be transferred to the top of the loop from two different locations, and the dual-source single-sink hyperedge represents that fact directly. Similarly, the single-source dual-sink hyperedge directly models the fact that control entering the top of the loop may proceed to either the body of the loop or to the statements following the loop. Furthermore, the SPG directly associates the condition controlling the flow of tokens along a branch with the branch that is affected. In most other graph representations, dataflow analysis is required to reveal the relationship between a condition and the path it controls.

The dashed boxes in Figure 4-3 depict groupings. The larger grouping represents the global scope of the program. The smaller grouping is used to indicate that the subgraph inside that grouping should be displayed in views as a single statement. This mechanism is used to compensate for the fact that statement granularity in an SPG may not match the statement granularity of a view displaying the system being represented. In this case, an SPG is unable to print a literal string, compute the value of an expression, and then print the value of that expression all in a single SPG node, so the statement has been broken down into its three constituent pieces, and the subgraph comprising those pieces has been

placed in a Statement grouping.

All but one of the edges in the SPG are control edges. The exception is the edge that runs from the node computing the value of sum/n to the node that writes out the value of d2. d2 is not a variable name, it is the name of an input port on the node. The node computing sum/n also has a named port (d1), but d1 is an output port. The data edge running from d1 to d2 indicates that tokens placed on d1 should be moved to d2. The value of the data to be placed on d1 is specified by making d1 the target of an assignment. Similarly, the use of the name d2 in the write statement indicates that the value to be written is the one carried by the token on that port.

This use of a data edge for moving a value from where it is computed to where it is used highlights another advantage of SPGs in comparison to the kinds of flow graphs usually used for Pascal-like languages. Conventional flow graphs typically employ an implicitly generated temporary variable to hold the result of a computation until that result is used, but such a strategy introduces an artifact into the underlying program representation: the representation contains a variable that does not exist in the original program. There is no need to introduce such spurious variables in an SPG, because an SPG can use data edges to directly express the fact that the result of a computation in one node is to be used in another node.

Annotations are depicted in Figure 4-3 as words in parentheses next to the SPG component to which they apply. For example, the node declaring the variables sum, n, and x contains the Start annotation, which indicates that execution of the SPG is to begin at that node. Similarly, the Stop annotation on the node that writes the value of d2 indicates that execution of the SPG should stop after that node has been executed. As the figure indicates, annotations are also used to specify the purpose for groupings.

The average program contains comments, and these can be easily represented in an SPG, but in order to simplify the presentation, I have omitted them from Figure 4-3. I provide a full description of how SPGs represent comments and other unexecutable information in Section 4.2.

Execution of an SPG is controlled by the flow of tokens through the graph. In general, tokens flow between nodes along edges, and nodes both consume and produce tokens. Nodes are eligible for execution whenever they have a token on each of their input ports, and during execution a node places tokens on its output ports. SPG execution is thus similar to execution of a variety of other graph structures, including Petri nets, flow graphs for dataflow languages [9, 8], and the Plan Calculus.

The topology of an SPG is essentially static during program execution, but the presence of conditions on destination branches means that the set of branches along which tokens may flow is determined dynamically. A token may only flow along a branch if the condition controlling that branch is true at the time the token arrives at the branch.

Simulated execution of the software system represented by an SPG is achieved by executing an interpreter that uses the SPG as its input. This interpreter is part of the *SPG Manager* (SPGM). I defer a full description of the SPGM until Chapter 6. For the purposes of this chapter, the SPGM can be thought of as interpreter for an SPG.

Interpretation of the SPG for the average program would proceed as follows:

1. The SPGM would search the SPG for the nodes with the Start annotation and would enable execution of those nodes. In this case, this would cause the node declaring the

variables sum, n, and x to be executed. A control token would then flow along the edge leading from that node to the node containing the assignment n := 0.

2. The assignment to n would be performed, and a token would flow to the node containing the assignment to sum. That assignment would be performed, and a token would flow to the node containing the read statement. The read into x would be performed, and a control token would flow to the empty node.

3. The empty node (essentially a null statement) would be executed, and a token would be placed on the edge emanating from the node.

4. The branch condition $x \neq -1$ would be evaluated, and if it were true, the token would follow that branch and arrive at the node incrementing n. Execution would then continue at Step 5, below. If the condition were false, the token would follow the other branch and would arrive at the node writing "Average is ". Execution would then continue at Step 6, below.

5. In a manner analogous to the straight-line code sequence at the beginning of the program, the assignment to n would be performed, then the assignment to sum, then another read into x would be performed. A token would then flow along the edge from the node performing the read to the empty node. Execution of the SPG would continue at Step 3, above.

6. The write would be performed, and a token would be placed on each of the edges emanating from the node. This would enable execution of the node with the assignment to d1, but would not enable execution of the node performing the second write, because that node would not be eligible for execution until it also received a token from the node performing the assignment to d1.

7. The assignment to d1 would be made by creating a new data token with the value sum/n and placing this token on the port named d1. The token would flow along the edge to the node containing the port d2. That node would then be eligible for execution, because it would have a token on each of its input ports.

8. The node performing the write of d2 would execute, causing the value of the token on port d2 to be written. The SPGM would notice that this node contained the annotation Stop, and interpretation of the SPG would cease.

As the remaining sections in this chapter will show, Semantic Program Graphs offer many more features than are apparent from this simple example. I defer a discussion of most of these features until they are formally introduced in the sections that describe them, but the SPG tasking mechanism crops up often enough to warrant a brief description here.

The SPG approach to tasking, i.e., to the simultaneous execution of multiple processes, each with its own program and data, is closely modeled on Ada tasks. SPG tasks are called STasks (simplified tasks — "S Tasks"), and STasks are implemented as a kind of SPG grouping, the STask grouping. Each STask may contain *entries* (modeled after Ada entries), and STask entries may be invoked by other STasks, just like subprograms may be invoked from different locations in a program. The statements making up an STask entry comprise an SPG subgraph, so entries, like subprograms, are also represented by groupings.

STasks and entries are used in the representation of software systems that deal with issues of mutual exclusion and interprocess synchronization and communication.

4.2 Unexecutable Components

SPG nodes, edges, and groupings may be either executable or unexecutable. Executable components have semantics associated with them that are fixed by the definition of an SPG. Unexecutable components have no such semantics. In this section I describe the unexecutable components of an SPG.

The unexecutable portion of an SPG is in some ways similar to an untyped hypermedia system: unexecutable nodes are connected to one another by unexecutable edges. Edges are not restricted to connecting to nodes; they may also connect to edges and groupings. This is useful when representing relationships involving edges and groupings. For example, an unexecutable grouping representing a file might be the source for an edge leading to a node that contains a comment describing the contents of the file. If a comment applied to more than one file, that relationship could be directly expressed by making each file grouping a source for the edge. This kind of n-to-1 relationship is much easier to express via SPG hyperedges than via the simple edges found in other graph-based program representations, because all the entities involved in the relationship are connected to the same edge.

Arbitrary annotations may be associated with nodes, edges, groupings, and annotations. An annotation is just a (name, value) pair, where both the name and the value are case-sensitive strings. The value of an annotation may be null. Annotations are thus similar in form to Unix and DOS environment variables, resources in the X window system, associative arrays in awk [2] and Perl [216], etc.

An unexecutable node with an annotation named Comment represents a program comment; the node contains the text of the comment. (The value of a Comment annotation is ignored.) Similarly, an unexecutable grouping with an annotation named File represents a file; the value of the annotation is the name of the file. The edge from the SPG components being commented on to the comment nodes is given an annotation named Has Comment, and the edge from the comment nodes to the SPG components being commented on is given an annotation named Comments On. Figure 4-4 depicts the unexecutable SPG components for a software system consisting of two files and a comment that applies to both of them. It should be noted that this is not a canonical form for such a system. An alternative representation would be to replace each hyperedge with two simple edges. For a further discussion of the issue of a canonical form, see Section 8.1.

Program comments are but one form of unexecutable information about software systems that can be expressed in SPGs. However, the annotation-based approach I have just described extends well to many different types of information. It is easy to imagine the representation of modification histories, project scheduling data, justifications for design and implementation decisions, etc. However, researchers into hypertext systems have expended considerable effort into the representation of this kind of information, and I was not interested in duplicating the efforts of that community in my development of SPGs. Instead, I focused my efforts on addressing the specific requirements of a representation for executable software systems in an environment supporting multiple editable views.

An example of such a requirement is the need to represent statement orderings within a program. In views based on the sequential control-flow paradigm, statement ordering

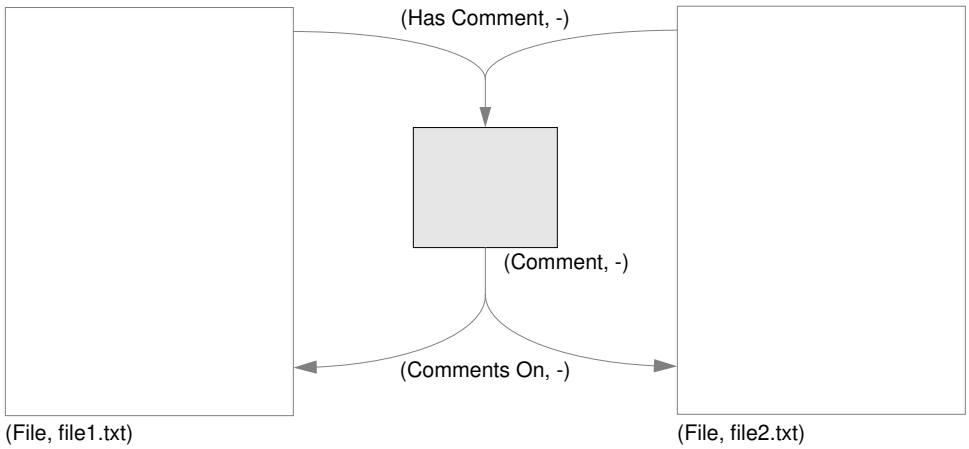


Figure 4-4: An SPG consisting of two unexecutable groupings and a single unexecutable node. Each grouping is annotated as a file, the node is annotated as a comment, and the edges between the groupings and the node are annotated as relationships between the comment and what is commented on. Null annotation values are indicated by a dash.

implies executable semantics: in the absence of explicit branching, the execution of statement n must immediately precede the execution of statement $n + 1$. For views arising from the dataflow and parallel control-flow paradigms, however, this is not necessarily the case. Consider the following two Occam commands (based on sample code in Ben-Ari's book [20, p. 108]), each of which consists of two statements to be executed in parallel:

```
PAR                              PAR
  south ! x                        east ? sum
  east ? sum                       south ! x
```

These two commands can be represented by the same SPG, because they have identical semantics. However, a programmer who writes the code on the left is unlikely to be satisfied with a view that displays it as shown on the right. This is an example of a situation not involving program commentary in which it is important for an SPG to be able to represent semantically neutral information about the depiction of a software system (in this case the fact that south ! x should precede east ? sum in views). SPGs use annotations to deal with such situations. For example, to address the problem of statement ordering, SPGs provide the Precedes annotation, which is attached to unexecutable edges and which imparts to views that the sources of the edge should be depicted prior to the destinations of the edge. (It is up to each view to determine what it means for one thing to "precede" another in the view. For conventional textual views, it almost certainly means that one thing appears earlier than another in the one-dimensional stream of text. For other views, especially two- and three-dimensional graphical views, the meaning of "precedes" is not necessarily as obvious.) A more complete discussion of how annotations can be used to represent unexecutable information about software systems is provided in Section 4-3.

Node Type	Purpose(s)
Declaration	Declare variables
Multiassignment	Perform computation, I/O, and variable assignments
Call	Call a subprogram or STask entry
Parameter	Pass parameter values non-strictly to a subprogram or STask entry
Accept	Accept a call to an STask entry if a call is pending
Epsilon	Explicitly do nothing; keep an SPG topologically valid

Table 4-2: Different types of executable nodes.

```
Perform the node's type-specific behavior.

(Remove input tokens)
For each executable input port on the node:
    Remove the token from the port and discard it.

(Put tokens on unused output ports)
For each executable output port on the node that did not receive a token during
performance of the node's type-specific behavior:
    If the port is a control port
        Create a new control token and place it on the port.
    Else (The port must be a data port)
        Create a new data token with the value UNDEFINED and place it on the port.
```

Figure 4-5: Approximate execution algorithm for nodes.

4.3 Executable Components

4.3.1 Nodes

There are six types of executable nodes in SPGs. Their functions are summarized in Figure 4-2.

Each executable node must contain at least one executable input port. A node becomes eligible for execution when it has tokens on each such port. An approximate execution algorithm for nodes is described in pseudocode in Figure 4-5. An exact algorithm is more elaborate and is given in Figure 4-39, but this simpler version suffices for the discussion that follows. Comments in the pseudocode are in parentheses and are italicized. The value UNDEFINED in Figure 4-5 has the computational properties of $\perp$.[1] Some of a node's behavior during execution is type-specific; a description of this type-specific behavior is described in the sections that follow for each type of node.

It is an error for more than one token to be placed on any output port of a node during an execution of that node.

[1] Thus, $x +$ UNDEFINED $=$ UNDEFINED for all x, and similarly for the other binary arithmetic operators. UNDEFINED is equal only to itself, so x is equal to UNDEFINED iff x has the value UNDEFINED.

Figure 4-6: Example Declaration nodes.

Declaration Nodes

Declaration nodes are used to declare variables. The contents of a declaration node is a comma-separated list of identifiers, each identifier being the name of a variable being declared. The list may be empty. The list may also contain duplicate entries, because references to a variable within an SPG are not in terms of the variable's name, but in terms of a unique integer identifier assigned to each variable by the SPGM. This integer is known as the *variable identifier* (VID) for the variable. As discussed in Section 3.5.4, uses of a variable within the SPG are in terms of its VID, not its name, and views are responsible for mapping from a VID to the appropriate name when generating presentations. Figure 4-6 shows some examples of Declaration nodes.

Because my research for this thesis did not encompass the modeling of data structures in software systems, all variables in an SPG represent integers and take on only integer values. The single exception is the value UNDEFINED, which may also be the value of a variable.

There is no type-specific execution behavior for Declaration nodes.

Multiassignment Nodes

Multiassignment nodes are where computation, input and output takes place. Each Multiassignment node contains a single multiassignment, the grammar for which is shown in Figure 4-7. This grammar offers a very restricted set of operators, but that is consistent with my limited concern for data manipulation in this thesis. Extending the grammar to support a wider variety of operators would be straightforward.

Each multiassignment consists of a comma-separated list of simple assignments. Assignment targets may be VIDs or the names of output ports. When a data output port is the target of an expression, a data token is created containing the value of the evaluated expression, and that token is placed on the output port. A control output port may only be the target of a null expression. In this case, a control token is placed on the port, but, because the SPGM places a token on any output port that does not receive one during the node's execution, it is rarely necessary to explicitly place a token on a control output port. In fact, the ability to explicitly place control tokens on Multiassignment output ports is provided only for compatibility with the capabilities of Call and Accept nodes, where the practice is both useful and necessary.

Expressions on the right-hand side of assignments may reference VIDs, the names of data input ports, or literal integer values. As shown in the grammar, VIDs are distinguished from literal values by preceding literal values with a "#" sign. When an input port is referenced in an expression, the value of the data token on that port is used in place of the reference. If there is no expression on the right-hand side of an assignment to a variable or a data port, the value of that expression is UNDEFINED.

Multiassignments support two distinct forms of parentheses. One form, syntactically indicated by conventional parentheses, is used only to preserve correct semantics: removal of such parentheses would change the meaning of an expression. Such parentheses are inserted

```
%token INTEGER IDENTIFIER STRING

%left '+' '-'
%left '*' '/'

%%

MultiAssignment:   Assignments | Nothing;

Assignments:       Assignment
                 | Assignments ',' Assignment
                 ;

Assignment:        Target ':=' Expression
                 | 'OUTPUT' ':=' STRING
                 | Target ':=' Nothing
                 ;

Nothing:         ;

Target:            Portname | VariableID | 'OUTPUT' ;

Portname:          IDENTIFIER ;

VariableID:        INTEGER ;

Expression:        Term
                 | Expression '+' Term
                 | Expression '-' Term
                 | 'INPUT'
                 ;

Term:              Factor
                 | Term '*' Factor
                 | Term '/' Factor
                 ;

Factor:            Portname
                 | VariableID
                 | '#' INTEGER
                 | '(' Expression ')'
                 | '$(' Expression ')$'
                 ;
```

Figure 4-7: A yacc grammar for Multiassignment nodes.

by views as they translate view-specific expressions into SPG representations of those expressions. These parentheses are absent in a view and should be suppressed by views when generating presentations for users. For example, in most languages, the expression a*b+c is equivalent to (a*b)+c, and this is how a*b+c would be evaluated in a Multiassignment node, too. However, in APL [70], operators are always evaluated right to left, so the APL expression a×b+c is evaluated as a×(b+c).[2] To ensure that the SPGM employs the correct semantics when evaluating this expression, the view translating from APL must insert parentheses, even though no parenthesis are present in the view.

The second form of parentheses in Multiassignment nodes is used to represent parentheses that *are* present in the original view, even if they are semantically unnecessary. Such parentheses are syntactically denoted by the form $(...)$. Both kinds of parentheses are treated the same when the SPGM evaluates expressions, but they are treated differently by views when mapping between program presentations and SPG representations of those programs.

Each Multiassignment node contains two implicit ports, a data input port called INPUT and a data output port called OUTPUT. These port names are reserved, so views may not create ports with these names. The INPUT port is considered to always contain a token, so that port can never limit the onset of execution of a Multiassignment node. During execution of an SPG, however, each time the value of INPUT is referenced during evaluation of an assignment expression, a datum is read from standard input, and that value is used as the value of the token on the INPUT port in that instance. If no data is available on the standard input, execution of the Multiassignment node blocks until data is available.[3]

When OUTPUT is the target of an assignment, the value of the expression being assigned is written to the standard output. If that value is UNDEFINED, the result of the write is undefined. If a Multiassignment node completes execution without assigning a value to OUTPUT, nothing is written to the standard output. As indicated by the grammar in Figure 4-7, strings may be written to the OUTPUT port. This provides a primitive facility for generating prompts, error messages, and the like. Strings in Multiassignment nodes, like the C strings after which they are loosely modeled, use the special sequence "\n" to indicate a newline.[4]

Pseudocode for the type-specific execution behavior for Multiassignment nodes is shown in Figure 4-8. As indicated by the pseudocode, the right-hand sides of all simple assignments in a multiassignment are evaluated before any assignments are made to targets. The order in which the different right-hand sides are evaluated is undefined. In fact, any number of the expressions may be evaluated concurrently. In this sense, evaluation of the component assignments making up a multiassignment is similar to the rules for evaluating assignments in CSP and function parameter lists in C.

Figure 4-9 shows some examples of Multiassignment nodes. It also introduces some conventions I use when drawing pictures of SPGs. First, input ports are drawn as triangles located on the top or the left-hand side of nodes, while output ports are depicted as triangles

[2] APL uses "×" for multiplication. The symbol "*" is used to indicate exponentiation in APL.

[3] SPGs currently have no concept of "end of file," but it would be easy to add this idea to SPGs. The most direct approach would be to define a distinguished data value, say, EOF.

[4] "\n" is the only special character sequence recognized — even "\\" and "\"" are unsupported. This is not because other special sequences are unimportant, but because they add no fundamental expressive power to SPGs.

```
(Evaluate right-hand side expressions)
For each right-hand side assignment expression:
    If (the expression refers to an invalid VID) or
      (the expression refers to the name of a control port) or
      (the expression refers to an input port name that doesn't exist)
        Signal an error.
    Else if the expression is null
        Set its value to UNDEFINED.
    Else
        Evaluate the expression.

(Make assignments to targets)
For each assignment target:
    If the target is a VID
        If the VID is invalid
            Signal an error.
        Else
            Assign the value of the right-hand side expression to the variable
            corresponding to the VID.
    Else (The target must be a port name)
        If there is no output port with that name
            Signal an error.
        Else if the port is a control port
            If the expression on the right-hand side is null
                Create a new control token and place it on the port.
            Else
                Signal an error.
        Else if the port name is OUTPUT
            Write the value of the right-hand side expression to the standard output.
        Else
            Create a new data token with the value of the right-hand side expression
            and place the token on the port.
```

Figure 4-8: Type-specific execution behavior for Multiassignment nodes.

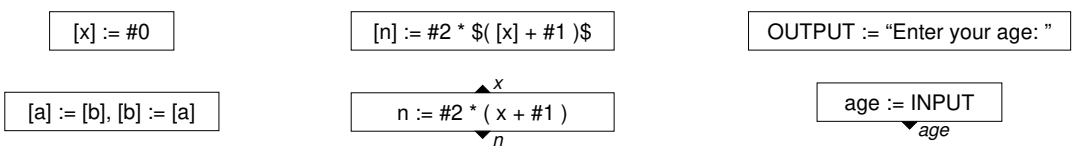


Figure 4-9: Example Multiassignment nodes. Left column: setting x to 0 and swapping the values of a and b. Middle column: two ways to set n to 2*(x+1), one based on references to variables, one based on data tokens. Right column: writing to the standard output and reading from the standard input.

located on the bottom or the right-hand side; the apex of input ports points up or left, while the apex of output ports points down or right. Ports for control tokens are unfilled, while ports for data tokens are filled. For example, the bottom node in the middle column of Figure 4-9 contains one data input port and one data output port. (An example of an SPG containing control ports can be found in Figure 4-43.) A second convention I employ in SPG depictions is that I enclose variable names in square brackets instead of using the corresponding VIDs. Although this is technically and syntactically incorrect, it makes it much easier to understand the examples.

Call Nodes

Call nodes are used to make calls to subprograms and to entries of STasks. In the discussion that follows, the distinction between a subprogram and an STask entry is usually unimportant, so I generally employ the neutral term *routine*, which should be understood to subsume both.

There are three components to a Call node: a specification of which routine to call, a specification of what tokens (typically representing parameter values) to pass to the routine, and a specification of what to do with tokens (typically representing results) returned from the routine. In most programming languages, routines are restricted to returning only a single value, but routines in an SPG are under no such limitation. This facility is necessary to support those languages which do allow multiple return values, e.g., Common Lisp. The grammars for the components of a Call node are shown in Figure 4-10.

Like variables, subprograms and STasks are referred to not by name, but by unique identifiers that are assigned by the SPGM: *routine identifiers* (RIDs) and *task identifiers* (TIDs), respectively. RIDs and TIDs are integers, so they can be used as data values for variables and tokens and can be passed around an SPG much as function pointers can be passed to and returned from functions in many languages.

The first part of a Call node specifies the values of the TID and RID corresponding to the routine to be invoked. The RID is mandatory, but the TID is optional. If a TID is specified, the called routine is an entry in the STask corresponding to the TID, and the RID identifies which entry is to be called. If the TID is omitted, the called routine is a subprogram, and the RID identifies which subprogram is to be called.

This design — that of specifying routines in terms of (TID, RID) pairs — is attractive for a number of reasons. First, it supports both subprogram and entry calls in a common framework, and this mimics the model of many programming languages that support both serial and parallel constructs. Such languages typically offer little syntactic distinction between intraprocess subprogram calls and interprocess communication;[5] this tends to be the case where callers of both subprograms and entries are anonymous, as in Ada, Occam, and systems featuring remote procedure calls [186].

A more important advantage of my design for call sites is that SPGs explicitly reflect the widespread outlook among software developers that sequential programs are in some sense fundamentally different from parallel programs. A Call node that omits the specification of a TID is naturally viewed as being part of a sequential program, because it does not

[5]This is often a deliberate design decision. Describing Ada, Ben-Ari notes that "Syntactically, an entry is exactly like a procedure declaration. This is done on purpose to allow substitution of a concurrent entry for a sequential procedure without otherwise changing a program" [20, p. 89].

```
Call:                TaskSpec RoutineSpec ;

TaskSpec:            'T' ':=' TaskID ',' | Nothing ;

RoutineSpec:         'R' ':=' RoutineID ;

TaskID:              PortName | VariableID | '#' INTEGER ;

RoutineID:           PortName | VariableID | '#' INTEGER ;

ParamValues:         ParamAssignments | Nothing ;

ParamAssignments:    ParamAssignment
                   | ParamAssignments ',' ParamAssignment
                   ;

ParamAssignment:     ParamSpec ':=' Expression
                   | ParamSpec ':=' Nothing
                   ;

ParamSpec:           'R' '.' PortName
                   | 'R' '.' '%' PortName
                   ;

Expression:          Term
                   | Expression '+' Term
                   | Expression '-' Term
                   ;

ReturnValues:        ReturnAssignments | Nothing ;

ReturnAssignments:   ReturnAssignment
                   | ReturnAssignments ',' ReturnAssignment
                   ;

ReturnAssignment:    Target  ':=' ParamSpec
                   | Nothing ':=' ParamSpec
                   | Target  ':=' 'RIID'
                   ;

Target:              Portname | VariableID ;
```

Figure 4-10: A yacc grammar for Call nodes. The nonterminals Call, ParamValues, and ReturnValues are the entry points for specifying which routine to call, what tokens to pass to the routine, and what to do with tokens returned from the routine, respectively. Tokens and nonterminals not defined in this grammar are the same as for Multiassignment nodes (see Figure 4-7).

even acknowledge the *existence* of other flows of control. An alternative design would have been to model sequential programs as degenerate concurrent programs with only a single process. The disadvantage of such an approach is that it would have made it more difficult for views to determine "what is going on" in a serial program, because, syntactically, every Call node might conceivably be invoking a routine in a different process.

STasks are groupings, so subprograms may exist both inside of and outside of STasks. Subprograms that are inside of an STask may only be invoked by Call nodes inside the same STask. This restriction reflects the fundamental need for encapsulation in higher-level IPC mechanisms like tasks and monitors. The ability to provide controlled access to shared resources (mutual exclusion) demands encapsulation, and the distinction between entries and subprograms reflects this demand: entries are visible outside an STask, subprograms are not.

Within a Call node, the routine to be invoked is known simply as R. As shown in Figure 4-10, this convention is part of the grammar. This simplifies the syntactic process of specifying how tokens should be passed from a call site to a routine and how tokens returned from a routine invocation should be processed at its call site.

The second part of a Call node specifies how information is to be conveyed from the call site to the routine invocation, i.e., how actual parameters map to formal parameters. It is similar to the body of a multiassignment, but the target of each assignment is a specification of an input port on the called routine that is to receive a token containing the value of the expression on the right-hand side of the assignment. Ports on the invoked routine are specified using the syntax R.*portname*.

I explain in Section 4.3.3 that routine ports usually have *two* names, one explicitly provided by the programmer at the time the routine is created, the other implicitly provided by the view (at the same time). At this point, it suffices to note the existence of the two different names and to remark that Call nodes may use either name when specifying ports on routines. However, any time a routine port is referenced, the reference must indicate which name is being used. Implicit names are used most frequently, and an unadorned port name is interpreted as an implicit name. Explicit names are preceded by a "%" sign. Details are provided in Section 4.3.3.

The final part of a Call node specifies how information is to be returned from the routine invocation to the call site, i.e., what happens to the routine's return values. It is a multiassignment-like construct where the right-hand side of each assignment is usually a specification of an output port on the called routine from which to extract the token whose value is to be assigned to the assignment target. It is also possible for the right-hand side of an assignment to consist of the special grammatical token RIID. This construct is used to facilitate non-strict parameter passing and is explained later when I describe Parameter nodes.

Figure 4-11 shows some example Call nodes. When drawing pictures of such nodes, I treat RIDs and TIDs in a manner analogous with VIDs, showing grouping names in square brackets instead of their corresponding (and technically required) identifiers.

The pseudocode for the type-specific execution behavior of Call nodes is shown in Figure 4-12. This pseudocode paraphrases some of the detailed tests that are explicitly present in the pseudocode for Multiassignment nodes. For example, the grammar for Call nodes indicates that TIDs and RIDs may be expressed as VIDs, port names, or literal values. The pseudocode in Figure 4-12 merely checks to see if a given TID or RID is "valid,"

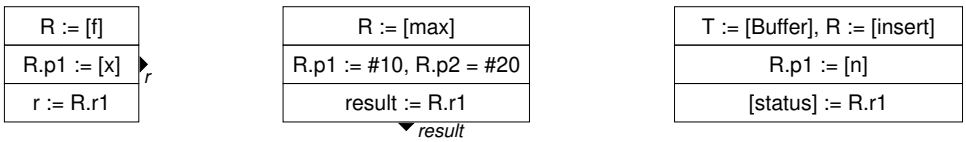


Figure 4-11: Example Call nodes. Left: a call to f with the actual parameter x. The value returned from f is placed on the Call node's output port r. Middle: a representation of the call, max(10,20). Right: a call to the entry insert in the STask Buffer, passing n as an actual parameter and receiving a status value in return.

but determining its validity requires establishing that a VID has a legitimate value; that a port name corresponds to one of the node's data input ports and that the value of the token on that port has a legitimate value; or that a literal value is a legitimate TID or RID. These details are omitted from the pseudocode. Also omitted are the tests to verify the validity of the right-hand side of expressions, but the same tests that apply to the right-hand sides of multiassignments also apply to the right-hand side of the multiassignment-like parts of Call nodes. Similarly, the pseudocode for Call nodes fails to rigidly distinguish between control and data ports, but the same "strong typing" restrictions apply for Call nodes as do for Multiassignment nodes (and indeed all nodes).

The primary difference between subprogram invocation and entry invocation is that subprogram invocation causes a copy of the invoked Subprogram grouping to be instantiated, and it is this new instantiation that is actually executed. No such instantiation is needed for STask entries, because it is illegal for an STask — itself the unit of mutual exclusion — to be executing more than one entry at a time.

As shown in Figure 4-12, when a Call node invokes an STask entry, it is suspended on a queue maintained by the SPGM for that STask. Execution of the Call node resumes when it is removed from the queue by an Accept node (see below). Execution resumes at the point immediately following the suspension code, so the next action taken by the Call node is the passing of input tokens to the called entry.

It is possible for a called routine to contain more output ports than are used in the third part of the calling node. That is, a Call node may choose to ignore some or all of the tokens produced by the called routine. (In terms of a concrete presentation, this is equivalent to a caller ignoring the return value of a routine it calls.) If a token is placed on an output port of a called routine and that port is not used on the right-hand side of any ReturnAssignment expression in the Call node making the call, the token is discarded.

The algorithm shown in Figure 4-12 corresponds to the strict invocation of a routine, because a Call node cannot commence execution until it has received all the tokens it will need to compute the actual parameter values to pass to the invoked routine. Non-strict routine invocations are also possible, but they require Parameter nodes in addition to Call nodes, as I describe below.

Parameter Nodes

Parameter nodes exist to allow tokens to be passed non-strictly to routine invocations. Such tokens typically carry data values representing actual parameters, but it is also possible to

```
If the TID is specified
    EntryCall ← TRUE. (This is an entry call)
Else
    EntryCall ← FALSE. (This is a subprogram call)

If (the TID is present but invalid) or (the RID is invalid)
    Signal an error
Else
    If EntryCall = TRUE
        Insert a marker representing this Call node into the queue for the STask
        corresponding to the TID specified in this Call node.
        Suspend execution of the Call node.
    Else (Instantiate a subprogram)
        Instantiate a new copy of the Subprogram grouping corresponding
        to the specified RID.

    (Pass input tokens to the routine)
    For each assignment in the ParamValues part of the grammar:
        Evaluate the right-hand side expression and place an appropriately valued token
        on the input port of the routine grouping that is the target of the assignment.

    (Collect return tokens from the routine)
    For each assignment in the ReturnValues part of the grammar:
        If the right-hand side of the assignment is other than "RIID"
            Wait until a token is placed on the routine output port that is the
            right-hand side of the assignment.
            If the target of the assignment is a valid VID
                Assign the value of the token to the variable corresponding to the VID and
                discard the token.
            Else if the target of the assignment is a valid output port name
                Place the token on the port.
            Else if the target of the assignment is null
                Discard the token.
            Else
                Signal an error.
        Else (The right-hand side of the assignment is "RIID")
            If the target of the assignment is a valid VID
                Assign the value of the invoked routine's RIID to the variable
                corresponding to the VID.
            Else if the target of the assignment is a valid data output port name
                Create a new data token with the invoked routine's RIID as its value and
                place the token on the port.
            Else
                Signal an error.
```

Figure 4-12: Type-specific execution behavior for Call nodes.

```
InvocationSpec:  'R' ':=' InvocationID ;

InvocationID:    PortName | VariableID ;
```

Figure 4-13: A yacc grammar for the first part of Parameter nodes. Nonterminals not defined in this grammar are the same as for Multiassignment nodes (see Figure 4-7).

pass control tokens to routine invocations.

Parameter nodes have a form that is very similar to that for Call nodes. The first part of a Parameter node specifies to which routine *invocation* the node applies. The second part is identical in form to the second part of a Call node, but for Parameter nodes, R refers to a particular routine invocation. The grammar for the first part of a Parameter node is shown in Figure 4-13.

Because SPGs can model systems exhibiting recursion and parallelism, it is possible for multiple invocations of a subprogram to be simultaneously active. Under such conditions, it is critical that tokens produced during execution of a subprogram be delivered to the correct call site. Similarly, for tokens that are delivered non-strictly to routine invocations, it is crucial that such tokens arrive at the correct invocation of the SPG subgraph corresponding to the routine that was called. To ensure that these needs are met, the SPGM dynamically assigns to each routine invocation a unique *Routine Invocation Identifier* (RIID). For Parameter nodes, the right-hand side of the assignment in the first part of the node is a value that is interpreted by the SPGM as an RIID. In practice, this value is always received from the Call node corresponding to the call site to which the Parameter node applies. Call nodes have access to the RIID for the current call through the special grammatical token RIID. If a Parameter node attempts to pass tokens to a routine invocation and that invocation no longer exists (see Section 4.5), the tokens are discarded.

The type-specific execution behavior for Parameter nodes is shown in Figure 4-14. The requirement that the final conditional statement be executed atomically is necessary to ensure that a routine invocation doesn't disappear between the time the conditional confirms its existence but before a token is placed on its input port.

Section 5.2.2 of this thesis contains examples of SPGs employing Call and Parameter nodes to implement strict, non-strict, and partially strict routine calls.

Accept Nodes

Accept nodes are used to accept calls to STask entries; they may only occur inside STask groupings. They are analogous in function to the Ada accept statement, after which they are named. The design of Accept nodes is complicated by the fact that although each of CSP-style messages, monitors, and Ada tasks require a rendezvous between the calling process and the called process, the duration of the rendezvous differs in each of these IPC models.

- For CSP, the caller and the callee rendezvous only long enough to effect a transfer of information from the caller to the callee; no values are returned to the caller. This unidirectional message-passing mechanism yields the shortest possible rendezvous.

```
If the RIID has never been issued by the SPGM
    Signal an error.
Else
    (Pass input tokens to the routine)
    For each assignment in the ParamValues part of the grammar:
        Evaluate the right-hand side expression.
        (The following conditional statement must be executed atomically)
        If the routine invocation corresponding to the RIID still exists
            Place an appropriately valued token on the input port of the routine
            invocation that is the target of the assignment.
```

Figure 4-14: Type-specific execution behavior for Parameter nodes.

- For monitors, all entries are procedures, and caller and callee rendezvous for the duration of the call to the monitor procedure. This is the longest possible rendezvous duration: the caller is blocked for as long as it takes for the monitor to fully service the call.
- For Ada tasks, the duration of the rendezvous is determined by the body of the **accept** statement handling the call. This may or may not contain all the code used to service the call, because code may follow the end of the **accept** body but still fall within the same branch of the **select** statement containing the **accept**.

As an example, consider the classic problem of controlling concurrent access to a bounded buffer. Two solutions to this problem are shown in Figures 4-15 and 4-16, each using a different IPC mechanism. The Ada solution is based on tasks, while the Concurrent Pascal solution is based on on the use of monitors. Because SPGs do not support buffer-like data structures (e.g., arrays), I have omitted from the figure the manipulations of the buffer itself, focusing instead on the control of concurrent access to this data structure.

In the Ada solution, the duration of the rendezvous is less than the full amount of time needed to service a call to **insert** or **remove**. This can be seen by the fact that **count** is incremented or decremented outside of the **accept** body but still inside the branch of the **select** containing the **accept**. In the monitor solution, however, manipulation of **count** must take place while the calling process is blocked, because code in a monitor (other than for initialization purposes) must be inside a monitor procedure.

In SPGs, the duration of a rendezvous is precisely the amount of time needed to fully execute the corresponding Accept node. The primary action of an Accept node is to execute the SPG subgraph contained inside a *Rendezvous grouping*. Structurally, a Rendezvous grouping is identical to a grouping for a subprogram, but a Rendezvous grouping can only be "invoked" by an Accept node. A Rendezvous grouping contains precisely those instructions which must be executed while the caller is blocked. If a Rendezvous grouping contains all the instructions needed to service an entry call, it acts like a monitor procedure. If it contains only some of the instructions needed to service a call, it acts like an Ada **accept** block. If the Rendezvous grouping is empty (in which case it may be omitted from the SPG), an entry call behaves like the sending of a CSP-style message.

```
task BBuffer is
  entry insert(item: in Integer);
  entry remove(item: out Integer);
end BBuffer;

task body BBuffer is
  count: Integer := 0;

  begin
    loop
      select when count < 10 =>
        accept insert(item: in Integer) do
          -- insert item into the buffer
          end insert;
        count := count + 1;
      or when count > 0 =>
        accept remove(item: out Integer) do
          -- remove an object from the buffer and assign it to item
          end remove;
        count := count - 1;
      end select;
    end loop;
  end BBuffer;
```

Figure 4-15: Solution to the Bounded Buffer problem using Ada tasks. This solution is based on one by Ben-Ari [20, pp. 89–92].

There are four components to an Accept node: a specification of the STask entry for which the node accepts calls; a specification of the Rendezvous grouping that is to be executed when the entry is accepted; a specification of how tokens passed to the accepted entry should be used; and a specification of how tokens to be returned from the entry call should be produced. The grammars for these four components are shown in Figure 4-17.

Execution of an Accept node proceeds as shown in Figure 4-18. In the same way that Call nodes refer to the routine to be invoked as R, Accept nodes refer to the Entry grouping for which calls are handled as E and to the Rendezvous grouping that determines rendezvous behavior as Z. The first two parts of an Accept node provide definitions for E and for Z.

Accept nodes (and, indeed, the SPG IPC machinery in general) would have a simpler structure if it were not necessary to employ two different groupings to handle entry calls, one to specify the interface to the entry (E), the other to specify the behavior of the entry (Z). Such a separation of concerns, however, is fundamental to IPC in Ada, where the instructions to be executed upon receipt of an entry call are specified on a per-select basis instead of on a per-entry basis. SPG support for IPC mirrors this separation so as to be able to accurately and directly represent Ada-style IPC (as well as other styles — see Section 5.2.4).

It is possible for an Accept node to omit the specification for the Rendezvous grouping. This is useful when modeling CSP-like IPC, because such IPC is based on a rendezvous that does nothing but transfer parameter values from caller to callee. These values can be

```
type BBuffer = monitor

var inserter, remover: queue;
    count: integer;

procedure entry insert(item: integer);
begin
  if count = 10 then delay(inserter);
  { insert item into the buffer }
  count := count + 1;
  continue(remover);
end;

procedure entry remove(var item: integer);
begin
  if count = 0 then delay(remover);
  { remove an object from the buffer and assign it to item }
  count := count - 1;
  continue(inserter);
end;

begin "initial statement"
  count := 0;
end
```

Figure 4-16: Solution to the Bounded Buffer problem using Concurrent Pascal monitors. This solution is based on one by Brinch-Hansen [27, p. 307].

saved in variables and/or passed onto output ports in the remaining two parts of the Accept node, so there is no need to delay the caller once the tokens have arrived at the Accept node. Hence, the instructions executed during the rendezvous are null, which is as it should be for simple message-passing.

The third part of an Accept node specifies what should happen to tokens passed to the entry. An entry acts like a shell around a subprogram: it takes parameters as input and it produces results as output, but it has no body. The body for the shell is determined by Accept nodes through their specification of which Rendezvous grouping should be executed when the entry call is accepted. For this reason, the most common action is to simply pass the tokens received by the entry onto the Rendezvous grouping actually handling the call. As noted above, however, it is also possible to store the values of tokens in variables and to pass them out of the Accept node through output ports.

The final part of an Accept node specifies what should happen to tokens produced by the Rendezvous grouping. As in the previous step, the most common course of action is to simply pass them on to the entry that was called (from where they will be forwarded to the caller), but they may also be stored in variables and placed on output ports of the Accept node.

Figures 5-5 and 5-6, which depict SPGs for the Ada and Concurrent Pascal solutions to the Bounded Buffer problem, show examples of Accept nodes in use.

```
EntrySpec:            'E' ':=' GroupingID ;

GroupingID:            VariableID | '#' INTEGER ;

RendezvousSpec:       'Z' ':=' GroupingID
                    | Nothing
                    ;

ParamTransfer:        ParamAssignments | Nothing ;

ParamAssignments:     ParamAssignment
                    | ParamAssignments ',' ParamAssignment
                    ;

ParamAssignment:      ZParamSpec ':=' EParamSpec
                    | ZParamSpec ':=' Nothing
                    | Target ':=' EParamSpec
                    ;

ZParamSpec:           'Z' '.' PortName
                    | 'Z' '.' '%' PortName
                    ;

EParamSpec:           'E' '.' PortName
                    | 'E' '.' '%' PortName
                    ;

ReturnTransfer:       ReturnAssignments | Nothing ;

ReturnAssignments:    ReturnAssignment
                    | ReturnAssignments',' ReturnAssignment
                    ;

ReturnAssignment:     EParamSpec ':=' ZParamSpec
                    | Nothing    ':=' ZParamSpec
                    | EParamSpec ':=' Target
                    | EParamSpec ':=' Nothing
                    ;
```

Figure 4-17: A yacc grammar for Accept nodes. The nonterminals EntrySpec, RendezvousSpec, ParamTransfer, and ReturnTransfer are the entry points for specifying which entry to accept a call for, what Rendezvous grouping should be executed when the call is accepted, what to do with tokens passed to the entry, and what tokens to place on the entry output ports, respectively. Tokens and nonterminals not defined in this grammar are the same as for Call nodes (see Figure 4-10).

```
(Determine the entry ("E") being handled)
If the RID for the entry handled by this Accept node is invalid
    Signal an error
Else
    E ← the RID for the entry grouping.
    Remove from the queue for the STask containing this Accept node the
    frontmost marker for a Call node invoking the entry E.
    Enable resumed execution of the Call node corresponding to the marker so removed.

(Determine the rendezvous grouping ("Z") to be executed)
If an RID for the rendezvous grouping is not specified
    Z ← UNDEFINED.
Else if the specified RID doesn't exist within the STask containing this Accept node
    Signal an error.
Else
    Z ← the RID for the rendezvous grouping.

(Handle input tokens)
For each assignment in the ParamTransfer part of the grammar:
    If (the right-hand side refers to an invalid port name) or
      (Z is UNDEFINED and is used on the left-hand side) or
      (the assignment target refers to an invalid port name or VID)
        Signal an error.
    Else if the target is a port
        Move the token from the right-hand side port to the target port.
    Else (The target is a VID)
        Assign to the variable corresponding to the target VID the value of the token
        on the right-hand side port.
        Discard the token on the right-hand side port.

(Handle output tokens)
For each assignment in the ReturnTransfer part of the grammar:
    If (the left-hand side refers to an invalid port name) or
      (the right-hand side refers to an invalid port name or VID) or
      (Z is UNDEFINED and is used on the right-hand side)
        Signal an error.
    Else if the right-hand side is a ZParamSpec
        Wait until a token is placed on the output port that is the right-hand side of the assignment.
        If the target of the assignment is not null
            Move the token from the right-hand side port to the target port.
        Else
            Remove the token from the right-hand side port and discard it.
    Else if the right-hand side is null
        Create a new token and put it on the specified target port.
    Else (The right-hand side must be a VID or a local port name)
        If the right-hand side specifies a port
            Move the token from the right-hand side port to the target port.
        Else
            Create a new data token and give it the value of the variable corresponding to
            the right-hand side VID.
            Place the token on the target port.
```

Figure 4-18: Type-specific execution behavior for Accept nodes.

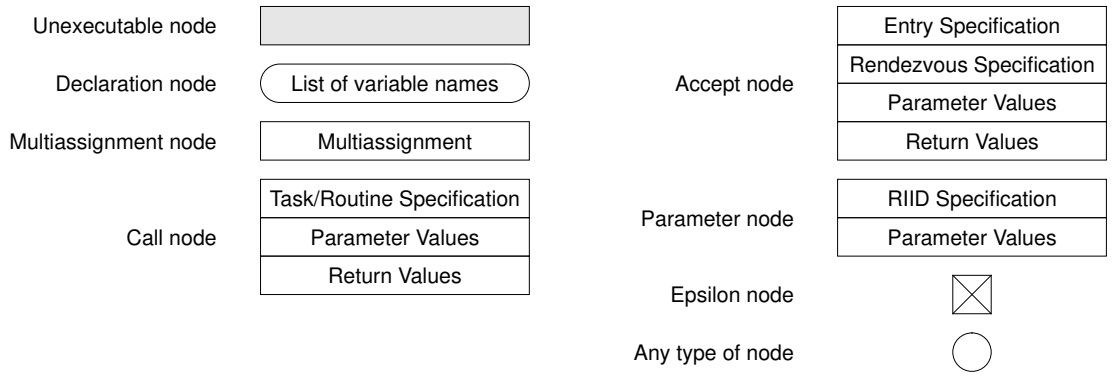


Figure 4-19: Graphical depictions of SPG node types.

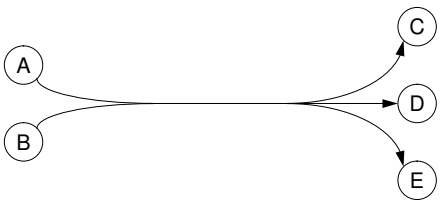


Figure 4-20: An executable edge along which tokens flow from A and B to C, D, and/or E.

Epsilon Nodes

Epsilon nodes are explicitly empty. Conceptually, they do nothing. Nonetheless, they can be extremely useful. First, they may be called upon to represent executable structures present in views that have no behavior. For example, a null statement in a programming language or a transition in a Petri net (see Appendix A) could be represented by an Epsilon node. Epsilon nodes can also be useful for synchronization purposes, as shown in Figure 4-21.

The second primary use for Epsilon nodes is to maintain the topological integrity of an SPG. An example of this kind of use can be seen in Figure 4-34, where an Epsilon node has been introduced to allow for the conceptual splitting of a single destination branch into two separate branches. By definition, Epsilon nodes that exist only to maintain the topological validity of an SPG do not correspond to any entity physically present in the software system being represented. As such, views are expected to elide them from presentations.

There is no type-specific execution behavior for Epsilon nodes.

Node Depictions

Each of these different node types has a characteristic graphical depiction that I use in figures for the remainder of this thesis. These graphical depictions are shown in Figure 4-19.

4.3.2 Edges

Figure 4-20 epitomizes an SPG edge. The semantics of this edge are determined by what happens when tokens from A or B arrive at the edge and by what happens to a token when it arrives at the point where the branches to C, D, and E depart. That is, the semantics of

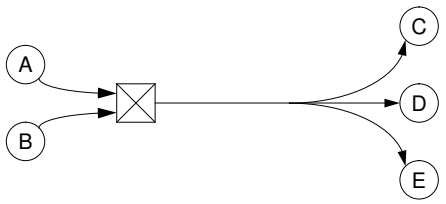


Figure 4-21: Tokens must be produced at both A and B before a token can flow from the Epsilon node to C, D, and/or E.

an edge are determined by the answers to the following questions:

1. What are the semantics of the junction where the source branches meet the edge body? (This junction is the *source junction*.)

2. What are the semantics of the junction where the destination branches meet the edge body? (This junction is the *destination junction*.)

The answer to the first of these questions is simple: a token arriving at the source junction simply proceeds to the destination junction. When an executable edge with more than one source is added to an SPG, the view adding the edge must ensure that execution of the SPG will never yield a state in which both A and B attempt to put tokens on the edge at the same time, because it is an error if more than one token is on an edge at any given time. In effect, the semantics of the source junction are *exclusive or*: either a token comes from A or a token comes from B, but a token must never come from both of them at the same time.

The exclusive or semantics for source junctions finds a natural application in my view model: it models the flow of computation coming to a common point after splitting at an earlier branch point. It implicitly asserts that a computation was carried out either at A or at B, but not at both places. An example of this use can be seen in Figure 4-25, which shows an SPG representation of a conditional construct.

Sometimes it is important to express the fact not that A *or* B must occur before an edge is traversed by a token, but that A *and* B must occur before the edge is traversed by a token. This can be expressed in SPGs by adding a Epsilon node to the graph, as shown in Figure 4-21. Because a node must have a token on each of its input ports before it is eligible for execution, the Epsilon node cannot produce a token on its output port until both A and B have placed tokens on their output ports.

This brings us to destination junctions. Of the possible semantics for a destination junction, these are three of the most useful:

- The token can be replicated and a copy put on each outgoing branch, thus leading to parallel flows of execution. Such behavior is necessary to meet my view model's requirement for dynamic process creation.

- One or more branches emanating from the junction can be nondeterministically chosen to receive copies of the token.

- The token can flow down a single branch that is deterministically selected on the basis of the program state. This behavior is needed to represent conditional branching.

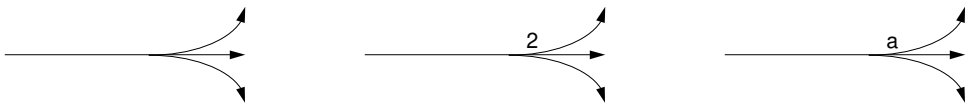


Figure 4-22: NP_1 edge (left), NP_2 edge (middle), NP_a edge (right).

In SPGs, these seemingly dissimilar capabilities are brought together in a single type of executable edge, the *nondeterministic parallel* (NP) edge.

NP Edges

The unification of parallel flow, nondeterministic flow, and deterministic conditional flow within an SPG is based on the generalization of nondeterminism from choosing 1 of n possibilities to that of choosing i of n (for $i \leq n$). If we call an NP_i edge one that nondeterministically chooses i of its n destination branches to receive copies of the token flowing along the edge, we can implement full parallelism through NP_n edges and we can implement conventional single-choice nondeterminism through NP_1 edges. Deterministic conditional execution can be represented by NP_1 edges with mutually exclusive conditions on the destination branches (see below).

It is often convenient to specify that an edge will put a copy of a token on each of its destination branches, regardless of the number of such branches. This is, for example, the semantics of Dijkstra's parbegin/parend construct [52] and Occam's PAR directive [20]. Such edges are called NP_a (*all*) edges. NP_a edges are more convenient for this purpose than are NP_n edges (for some fixed n), because the number of destination branches emanating from an edge may change as an SPG is edited. NP_n edges tightly couple the semantics of the edge with the current number of destination branches, but with NP_a edges, the coupling is avoided.

There is thus only one type of NP edge, the NP_i edge, with the special case that NP_a means that i has as its value the current number of destination branches. In those cases where i is unimportant, NP_i edges are referred to as NP edges. In this thesis, NP_i edges are graphically depicted with the value of i near the destination junction. The case of NP_1 is so common that it is the default: a destination junction without a value of i explicitly specified indicates that the edge containing the junction is an NP_1 edge. Figure 4-22 gives examples of these conventions.

Associated with each destination branch of an NP edge is a condition that controls whether that branch may be traversed by a token at that time. A condition is a boolean expression based on the grammar for arithmetic expressions defined for Multiassignment nodes. It differs only in its additional support for the standard logical and relational operators, the syntax and semantics for which I borrowed from C. The grammar for conditions is shown in Figure 4-23.

There is some tension in the design of the grammar for branch conditions. If it is too restrictive, it will not be possible for views to directly represent common conditionals such as if x + y > z, and that limitation would obscure "what is going on." On the other hand, if it is too expressive, views will be forced to expend significant effort to parse branch conditions on top of having to make sense of the SPG as a whole. I have therefore chosen a middle ground, whereby the expressiveness of the grammar for branch conditions is tied to

```
Condition:     LogicalExp
             | LogicalExp '&&' LogicalExp
             | LogicalExp '||' LogicalExp
             | '!' LogicalExp
             | 'default'
             | Nothing
             ;

Expression:    Term
             | Expression '+' Term
             | Expression '-' Term
             ;

Factor:        Portname
             | VariableID
             | '#' INTEGER
             | '(' LogicalExp ')'
             | '$(' LogicalExp ')$'
             ;

LogicalExp:    Expression '<' Expression
             | Expression '<=' Expression
             | Expression '==' Expression
             | Expression '!=' Expression
             | Expression '>=' Expression
             | Expression '>' Expression
             ;
```

Figure 4-23: A yacc grammar for conditions on destination branches. Tokens and non-terminals not defined in this grammar are the same as for Multiassignment nodes (see Figure 4-7).

the expressiveness of Multiassignment nodes. Given that SPG views must contend with the expressions inside Multiassignment nodes anyway, it is not an onerous burden to require that they also be able to cope with the three standard logical and the six standard relational operators. Note, however, that function calls are not part of the grammar for branch conditions, so conditions such as if $f(x) \geq 0$ must be expressed in terms of a call node (for the call to $f(x)$) that delivers a data token representing the result of the call to the branch condition dependent on that result.

To represent conventional branching constructs, i.e., sequential conditional execution, views construct an SPG subgraph consisting of an NP_1 edge with two or more destination branches. Each branch is controlled by a condition, and the conditions are designed such that exactly one is true any time a token is at the destination branch.

There are two special conditions. First, a null condition is defined to be true, so branches lacking explicit conditions are treated as branches with conditions that always evaluate to true. In conjunction with the convention that unlabeled destination junctions are implicitly of type NP_1, this allows flow graphs for straight-line code to be drawn in the traditional

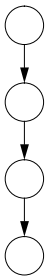


Figure 4-24: SPG for straight-line code. Each edge has a single destination branch emanating from a destination junction that is implicitly of type NP_1. This destination branch has a null condition, hence is controlled by an implicit condition that always evaluates to true.

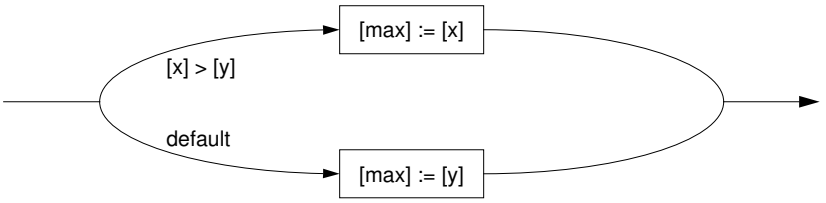


Figure 4-25: SPG for a conditional statement.

manner; see Figure 4-24. The other special condition is the distinguished expression default, which is defined to be true iff the conditions on all the other branches leaving the destination junction have already evaluated to false. The default condition is useful for modeling else branches of if-then-else constructs. It also naturally models the default label in C's switch statement and the others label in Ada's case statement. No more than one destination branch on an edge may be controlled by the default condition.

As an example of how NP edges can be used, Figure 4-25 shows an SPG representation of the following Pascal statement:

```
if x > y then max := x else max := y;
```

The SPG clearly indicates that one branch of the conditional is under control of the condition x > y, while the other branch is under control of the special condition default.

Conditions on destination branches are like Multiassignment nodes in that they may contain input ports at which other destination branches terminate. Unlike Multiassignment nodes, branch conditions may not contain executable output ports (although they may contain unexecutable output ports). After evaluation, all tokens used in the evaluation of a branch condition are discarded.

An NP edge is eligible for execution when each of the following conditions is fulfilled:

- A token is present at the edge's destination junction.
- A token exists on each executable input port of each branch condition that is part of the edge.

Execution of an NP edge proceeds according to the algorithm shown in Figure 4-26. An example of a branch condition that uses input tokens can be found in Figure 4-34.

```
(Determine the set of branches with true conditions)
EligibleBranches ← ∅.
For each destination branch in the edge with a condition other than "default":
    If the branch's condition refers to an invalid port name or VID
        Signal an error.
    Else if the branch has no condition
        Add the branch to EligibleBranches. (Implicit conditions are always true)
    Else
        Evaluate the branch's condition.
        If the condition is true
            Add the branch to EligibleBranches.

(See if the default branch should be taken)
If (the edge contains a destination branch with the condition "default") and
  (EligibleBranches is the empty set)
    Add the "default" branch to EligibleBranches.

(Move a copy of the token at the destination junction along some eligible branches)
TrueBranches ← cardinality of EligibleBranches.
Randomly choose min(i, TrueBranches) of the branches in EligibleBranches.
For each branch so chosen:
    Make a copy of the token at the edge's destination junction and
    place it on the input port where the branch terminates.

(Remove all tokens from the edge)
Remove the token from the edge's destination junction and discard it.
For each destination branch in the edge:
    For each executable input port on the branch condition:
        Remove the token from the port and discard it.
```

Figure 4-26: Execution algorithm for NP_i edges.

In view of the similarity between branch conditions and Multiassignment nodes, it is reasonable to ask why a condition isn't simply a special type of node. The primary reason is that nodes and branch conditions serve different purposes. The main function of a node is to prepare for or to carry out a computation. The main function of a branch condition is to control token flow. Nodes can produce tokens that flow to other nodes and are used in other computations. Conditions cannot. Conditions can prevent tokens from flowing along an edge. Nodes cannot. Maintenance of this distinction between computation for its own sake (nodes) and computation for the purpose of controlling conditional execution (conditions) is crucial if a CR is to preserve a sense of "what is going on." Edges represent possible paths of computation flow, and branch conditions dynamically determine which paths can be taken. As such, the condition controlling a path is an essential part of the path itself, and the SPG representation of branch conditions directly reflects this.

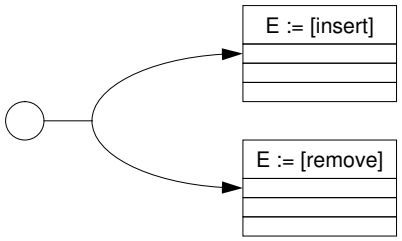


Figure 4-27: NP edge leading to two Accept nodes. Branch conditions are irrelevant and have been omitted.

Select Edges

NP edges suffice in most contexts, but they are inappropriate for use with Accept nodes, because an Accept node cannot sensibly execute unless a call to the entry it handles is pending. For example, consider again the Ada task for managing a bounded buffer (Figure 4-15). When the buffer is neither full nor empty, the task will accept a call to either insert or remove. Using an NP edge, that fact would be represented as shown in Figure 4-27. The problem with this representation is that when a token arrives at the destination junction of the edge, the branch along which it flows is chosen nondeterministically *without taking into account* whether a call is pending for the entry handled by the Accept node to which it flows. As a result, it is possible that the token would flow to the Accept node handling the insert entry when all pending calls were for remove (or vice versa). Similarly, if no calls were pending, the token might flow to the Accept for remove, but the next entry call might be to insert. Such behavior is both inherently unreasonable and inconsistent with the semantics of guarded nondeterminism in languages like Ada, Concurrent Pascal, and CSP.

To handle this problem, SPGs offer a second type of executable edge: Select edges. Like their Ada namesakes, Select edges and Accept nodes work together to implement a wide range of IPC mechanisms. Their interaction affects the structure and behavior of SPGs in the following ways:

- Like Accept nodes, Select edges may occur only inside STask groupings. Only Select edges may contain destination branches leading to Accept nodes, and, with one exception, destination branches in Select edges may lead only to Accept nodes. The single exception is that a Select edge may contain at most one destination branch leading to a non-Accept node, but *only* if the condition on that branch is default. This provision allows SPGs to represent software systems that do not block when some form of IPC is expected, but none is pending. This behavior is exhibited, for example, by Ada's select-else construct.

- Each destination branch of a Select edge that leads to an Accept node has an implicit additional conjunct as part of that branch's condition. The value of this conjunct is true iff a call is pending to the entry handled by the Accept node at the time the

branch condition is evaluated.[6]

- If *all* destination branches of a Select edge lead to Accept nodes, all branch conditions are false, and at least one branch condition would have been true at the time of evaluation had one or more entries had calls pending, the token at the destination junction remains at the destination junction until a call is made to an entry handled by one of the Accept nodes to which the edge leads. When such a call is made, the token is moved from the destination junction to an appropriate Accept node.

A Select edge is eligible for execution under the same conditions as an NP edge: when (1) a token is present at the destination junction of the edge, and (2) each of the edge's branch conditions can be evaluated. The execution behavior of Select edges is specified in Figure 4-28.

I considered leaving Select edges out of SPGs, because, in the common case where the Select edge has a single source and where all destination branches of the edge lead to Accept nodes, the behavior of the Select edge can be approximated by the use of an NP edge with an additional branch controlled by the condition default. This additional branch simply leads back to the node from which the edge emanates, leading to an infinite loop that can only be escaped by passing control to an Accept node. This approximation is shown in Figure 4-29, which also demonstrates that in this thesis I use thick lines to depict Select edges and thin lines to depict NP edges.

The problem with this approximation is that it repeatedly tests the conditions controlling token flow along the destination branches, and that is contrary to the semantics of most IPC guards. For example, consider again the code in Figure 4-15. The semantics of Ada dictate that the when conditions inside the select statement are to be evaluated only once — when control enters the select statement. Even if the value of count changes as the select statement awaits a call to insert or remove, the value of the when expressions will not be recomputed. This kind of behavior cannot be faithfully represented using an NP edge. It is, however, directly expressible through the use of a Select edge.

4.3.3 Groupings

There are four different kinds of executable groupings: Subprogram groupings (and their instantiations), STask groupings, Entry groupings, and Rendezvous groupings. In contrast to the situation with nodes and edges, however, it is not uncommon for a grouping to play more than one role. For example, a monitor procedure may be represented by a grouping that is both an entry and a rendezvous. Furthermore, there is often a tight coupling between one or more of these kinds of grouping and other (unexecutable) groupings that correspond to the same region of a program. For example, it is common for a subprogram to also be a scope, but this is not always the case (witness the original BASIC, where all variables are global). Similarly, in languages like C and C++, a grouping corresponding to a file would comprise a scope, but in languages like Pascal and Ada, a file does *not* comprise a scope.

[6] For Select edge branches leading to Accept nodes, then, it must always be possible to determine which entry is being handled by the Accept node. If this were not the case — for example, if the RID of the entry could be passed into an Accept node through a data port — it might be impossible to determine the value of the condition on such a branch. It is to prevent such anomalies that GroupingIDs in the grammar of Figure 4-17 are restricted to being either variable values or literal values.

```
(Determine the set of branches with true conditions)
EligibleBranches ← ∅.
For each destination branch in the edge with a condition other than "default":
    If the branch's condition refers to an invalid port name or VID
        Signal an error.
    Else if the branch has no condition
        Add the branch to EligibleBranches. (Implicit conditions are always true)
    Else
        Evaluate the branch's condition.
        If the condition is true
            Add the branch to EligibleBranches.

(Determine the set of branches waiting for a new entry call or for an existing call to terminate)
WaitingBranches ← ∅.
For each branch in EligibleBranches:
    If (the branch leads to an Accept node) and
      ((no call is pending for the entry handled by that Accept node) or
       (a call is pending for that entry but the entry is already executing))
       Remove the branch from EligibleBranches and add it to WaitingBranches.

(See if the default branch should be taken)
If (the edge contains a destination branch with the condition "default") and
  (EligibleBranches is the empty set)
    Add the "default" branch to EligibleBranches.

(If possible, move copies of the token on the edge along the eligible branches)
If the cardinality of EligibleBranches > 0
    Randomly choose one of the branches in EligibleBranches.
    Make a copy of the token at the edge's destination junction and
    place it on the input port where the chosen branch terminates.

(If necessary, wait for an entry call)
Else
    if the cardinality of WaitingBranches > 0
        Wait until (a marker for a call to an entry handled by an Accept node led to by
                      a branch in WaitingBranches is in the queue for the STask
                      containing this edge) and
                   (the entry corresponding to that call marker is not currently being executed).
        Randomly choose one of the branches in WaitingBranches leading to an Accept node
        handling the entry corresponding to the call marker.
        Make a copy of the token at the edge's destination junction and
        place it on the input port where the chosen branch terminates.

(Remove all tokens from the edge)
Remove the token from the edge's destination junction and discard it.
For each destination branch in the edge:
    For each executable input port on the branch condition:
        Remove the token from the port and discard it.
```

Figure 4-28: Execution algorithm for Select Edges. Any reference to a branch condition in this pseudocode is to the *explicit* branch condition, i.e., to the branch condition before augmentation with the implicit conjunct indicating whether a call is pending for the Accept node to which a branch may lead.

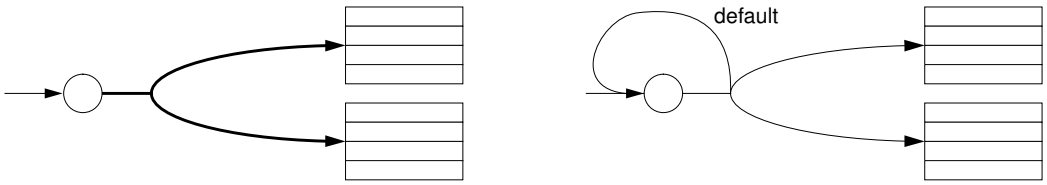


Figure 4-29: Emulating a Select edge (left) with an NP edge (right).

This lack of uniformity in the world of programming languages makes it convenient to treat the characteristics corresponding to each kind of grouping as independent attributes. As a result, SPGs have but a single type of executable grouping. The attributes that determine the precise semantics of an executable grouping are specified using standard annotations. Hence, subprograms are represented by executable groupings with an annotation named Subprogram, task entries by executable groupings with an annotation named Entry, monitor procedures by executable groupings with an annotation named Entry and a separate annotation named Rendezvous, etc. A complete list of these standard annotations can be found in Table 4-3.

An alternative design would have been to create four different types of executable groupings, one each for subprograms, tasks, entries, and rendezvous. Given such a design, the fact that an SPG subgraph represented more than one thing (e.g., both an entry and a rendezvous) would be indicated by containing the subgraph inside more than one grouping. This approach is straightforward in concept, but there are some troublesome subtleties that must not be overlooked in practice. For example, it must be the case that each of the independent groupings has exactly the same contents, which means, among other things, that none of the groupings may contain any of the others. More significantly, the contents of the different groupings would have to remain consistent as the subgraph was edited, so, for example, if a node were added to a subgraph that was both a subprogram and a scope, the view adding the node would have to ensure that the node was added to both groupings. Finally, views would have to have a way to efficiently determine whether any arbitrary pair of groupings had exactly the same contents, because this would tend to indicate that the two groupings corresponded to a single semantic object; this would be important information for the generation of effective presentations. Each of these drawbacks is absent in the annotation-based approach I adopted for SPGs.

For the purpose of verifying the validity of an SPG, each grouping must independently satisfy the constraints for each of its roles. For example, a Rendezvous grouping must be contained inside an STask grouping, but a Subprogram grouping has no such restriction. A grouping that is both a subprogram and a rendezvous, then, must be contained within an STask, because otherwise the grouping is invalid in its role as a Rendezvous grouping. In general, if a grouping plays more than one role, it must satisfy the *union* of the constraints that apply to each of the individual roles it plays.

Within this thesis, I graphically depict groupings as shown in Figure 4-30.

Subprogram Groupings

Subprogram groupings are used to represent subprograms, i.e., conventional procedures and functions as used in serial programming languages. When depicting Subprogram groupings,

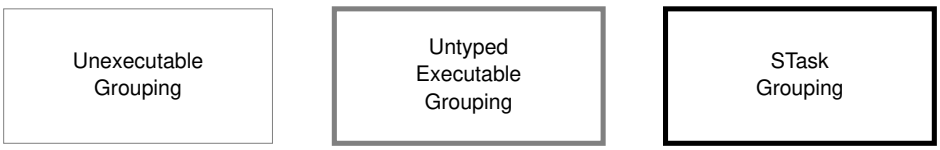


Figure 4-30: Graphical depictions of unexecutable groupings, executable groupings, and STask groupings. An STask grouping is an executable grouping with an annotation named STask.

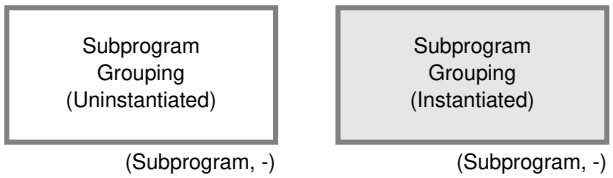


Figure 4-31: Graphical depictions of an uninstantiated Subprogram grouping (i.e., the static "source code" for a subprogram) and an instantiated Subprogram groupings (i.e., a runtime subprogram invocation).

it is convenient to be able to distinguish between a static Subprogram grouping itself and its dynamic instantiations. I use the depictions shown in Figure 4-31.

Executable ports on Subprogram groupings serve as specifications of tokens to be passed into and out of subprograms. Both control and data tokens may be so passed. Data tokens correspond to formal parameters and formal results for a subprogram; they are used in the representation of both control flow-based programs and dataflow-based programs. Control tokens represent a transfer of control from a call site to a subprogram and from a subprogram to a call site; they are employed only in the representation of software systems based on control flow.

Call nodes and Parameter nodes use subprogram port names to specify how tokens flow to and from Subprogram groupings.[7] This is adequate for most calls, in particular, for calls where the caller knows the names of the callee's ports. However, RIDs may be passed around an SPG as data values, so it is possible for a caller to know only the RID of a routine and the number of that routine's parameters and results; the port names of the routine are unknown to the caller. For example, consider this C function:

```
int indirectCall(int (*pf)(int), int arg)
{
  return (*pf)(arg);
}
```

Clearly indirectCall cannot know the name of the formal parameter expected by the function *pf, yet just as clearly it has no difficulty in making a valid call. The reason for this is that the correspondence between actual and formal parameters in most languages is determined from *positional* information: the position of an expression in a call's argument

[7] Ports on SPG components need not be named, but if a port does have a name, that name must be unique for that component. A name used in one component, however, may be reused for a port on a different SPG component.

list determines the parameter to which it maps in the called routine's formal parameter list. In effect, this positional information yields an *implicit name* for each formal parameter. The result is that subprogram parameters really have two names: the explicit name provided by the programmer, and the implicit name derived from the parameter's position in the parameter list. Some languages — for example, Ada and Common Lisp — allow arguments to be mapped to parameters using either or both of these sets of names. That is, call sites in these languages may pass parameters positionally and/or in terms of the names of the formal parameters.

SPGs don't use parameter lists for passing tokens to and from subprograms, because not all languages employ parameter lists. In particular, graphical languages often employ a plug-and-socket metaphor for associating actual parameters with formals. In such languages, there is no implicit name for a parameter based on its position in a parameter list, because there is no parameter list. Nonetheless, the function indirectCall, above, demonstrates the usefulness of being able to decouple knowledge of a subprogram's formal parameter and result names from the ability to invoke that subprogram.

The solution I adopt in SPGs is to explicitly recognize that ports on subprograms may have two names. One name is the conventional user-assigned name for a parameter or result. This is the name that typically shows up in the concrete syntax of a view presentation. The second name is silently generated by the view that creates the port. This second name is of the form p*n* (*parameter n*) for grouping input ports and r*n* (*result n*) for grouping output ports, with n in each case indicating the position of the port in a canonical ordering of the input and output ports, respectively. The canonical ordering for data ports starts with $n = 1$ and is initially established by the view that creates the Subprogram grouping. The ordering is updated by views that add new executable ports to and/or remove existing executable ports from the grouping. The positional names p0 and r0 are reserved for the input and output control ports corresponding to the explicit transfer of control to the subprogram from the call site and from the subprogram to the call site, respectively. Hence, views can rely on the fact that all subprograms invoked via control flow will have a control input port p0 and a control output port r0.

Call nodes and Parameter nodes that reference ports on Subprogram groupings may use either the user-defined name or the positional name for a given port. User-defined names (i.e., conventional parameter names) are represented in an SPG via an annotation named Name, while view-assigned names (i.e., canonical ordering information) are represented via an annotation named PR Name (*Parameter-Result Name*). In both cases, the value of the annotation is the port's name. Within a Call or Parameter node, a reference to a port in terms of its PR name simply uses the name itself. A reference to a port in terms of its user-defined name, on the other hand, precedes the name with a percent sign ("%").

As an example, consider the following Ada function and two calls to it:

```
function max(value1, value2: in integer) return integer;

max(10, 20);                          -- use positional names
max(value1 => 10, value2 => 20);      -- use user-defined names
```

Figure 4-32 shows how this function and these calls can be modeled using SPGs. Note the use of assignments involving ports p0 and r0 to achieve synchronization between caller and callee.

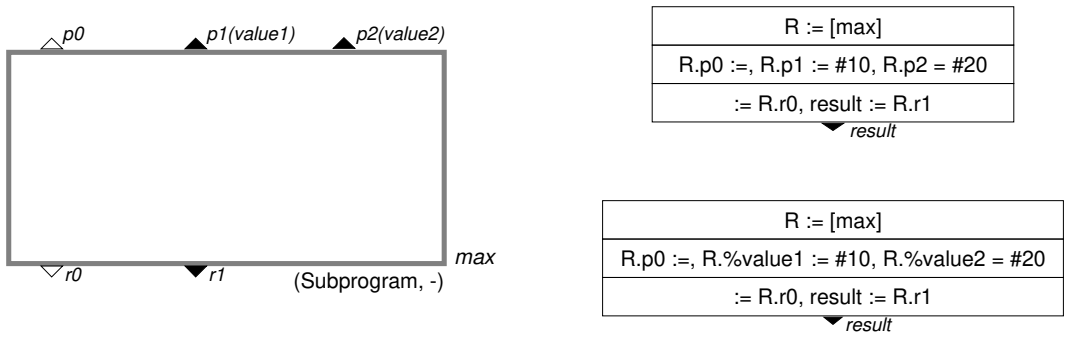


Figure 4-32: SPG representation for the Ada **max** function (left) and calls to it using positional (upper right) and user-defined (lower right) parameter names.

I noted above that names for SPG components are stored as annotations, but it is convenient to use a shorthand notation for names when drawing pictures of SPGs. Names are therefore italicized and written near the component to which they apply. For executable ports on Subprogram groupings, both PR names and user-defined names are shown, with PR names preceding user-defined names and user-defined names in parentheses. In the SPG in Figure 4-32, the grouping is named "max" and its input ports have PR names "p0," "p1" and "p2;" the latter two ports also have user-defined names "value1" and "value2," respectively. The grouping's output ports only have PR names: "r0" and "r1." Each of these names would of course be represented using an annotation in the actual SPG.

The executable ports of a Subprogram grouping are in some sense "virtual," because the input ports do not control eligibility for execution of the grouping, in contrast with the role played by input ports on executable nodes. Rather, they exist to facilitate abstraction of subprograms, i.e., the ability of views to depict a subprogram by presenting only its external interface. This allows subprograms to be displayed like nodes even though they behave like groupings. This feature is important, not only for views employing abstraction, but also because SPGs do not offer "non-strict" execution of nodes (a node cannot execute until it has a token on of its each input ports) but SPGs do support non-strict subprogram invocations (a subprogram may be invoked before tokens are available for each of its input ports).

Executable ports on subprogram groupings have a dual nature that arises from their role in supporting abstraction. An input port on the external interface of a subprogram acts like an output port when viewed from within the subprogram, because it is a portal through which tokens arrive from outside the grouping. Similarly, what appears to a caller to be an output port on a Subprogram grouping acts like an input port when viewed from within the subprogram, because it is a sink for result tokens produced by the subprogram. SPGs thus offer two specialized port types that are applicable only to executable groupings: *InOut* ports are for tokens entering the grouping, and *OutIn* are for tokens leaving the grouping. These names reflect the fact that InOut ports behave like input ports from outside the grouping, but behave like output ports from inside the grouping; similar reasoning applies to OutIn ports. Technically, then, the executable ports on executable groupings are not input or output ports, but are instead InOut and OutIn ports. Informally, however, it is often convenient to refer to the former as input ports and to the latter as output ports, and

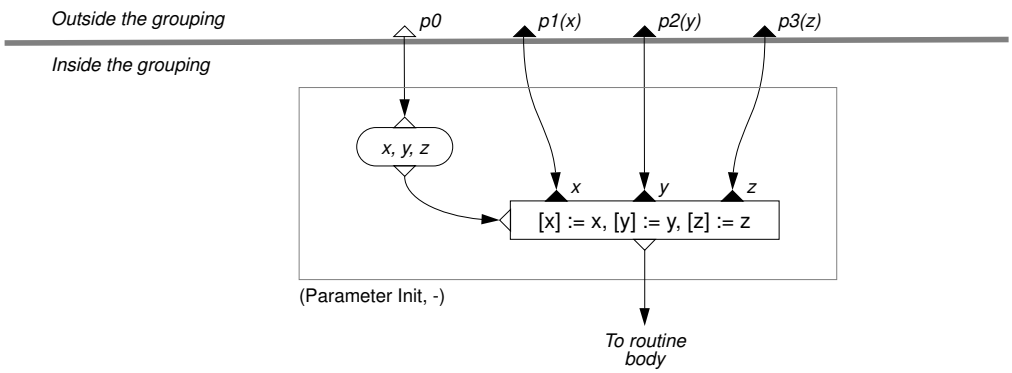


Figure 4-33: Using a Parameter Init grouping to initialize local parameter variables.

that is in fact what I have done up to this point in this dissertation.

A token arriving at an InOut port on a Subprogram grouping (which can only result from execution of a Call or Parameter node) is immediately placed on the branch within the grouping emanating from the port. If there is no such branch, the token is discarded. A token arriving at an OutIn port (which can only result from token flow within the subprogram graph) is immediately forwarded to the Call node that invoked the subprogram. If there is no multiassignment in the result part of the Call node waiting for a token from that port, the token is discarded.

Data InOut ports on subprogram groupings correspond to formal parameter declarations, but because each parameter name is associated only with a port, there is no way to use the formal parameter name within the body of the subprogram as if it were a local variable. For example, it is not possible to change the value of a token on a data InOut port by making it the target of an assignment. Dataflow languages don't usually allow assignments, so this is not a problem for them, but it is a serious impediment to the accurate representation of languages based on control flow.

I address this problem in SPGs by providing a mechanism to declare local variables that represent a subprogram's formal parameters, allowing such variables to be initialized with actual parameter values when the subprogram is invoked, and clearly identifying instances of such parameter declaration and initialization. Each facet of this solution has a corresponding SPG realization: local variables are declared in a Declaration node, initial values are assigned in a Multiassignment node, and the SPG subgraph corresponding to parameter declaration and initialization is enclosed in a grouping with an annotation named Parameter Init. Figure 4-33 shows how these pieces fit together for a control-flow based subprogram that is called strictly with three formal parameters, x, y, and z.

The Parameter Init grouping contains a Declaration node and a Multiassignment node. For each formal parameter represented by a data InOut port on the Subprogram grouping, the Declaration node contains a declaration for a variable of the same (user-defined) name, and the Multiassignment node contains an assignment that initializes the variable with the value of the token that is passed onto the corresponding port. Control edges lead from InOut port p0 to the Declaration node, from the Declaration node to the Multiassignment node, and from the Multiassignment node to the beginning of the body of the Subprogram grouping. This ensures that variables representing formal parameters are declared and

initialized before they can be used. In addition, data edges lead from the data InOut ports to the Multiassignment node, thereby delivering the necessary initialization values to the node where the variables can be initialized.

Because Subprogram groupings are *templates* from which executable portions of the SPG are dynamically instantiated, no executable edges may cross the boundary of a Subprogram: executable edges within an SPG are either entirely within a Subprogram grouping (in which case they are part of the template) or entirely outside of a Subprogram grouping (in which case they do not interact with the subprogram). For unexecutable edges, however, there is no such restriction, because unexecutable edges have no executable semantics. In Section 4.5, where I describe the process of SPG execution, I explain what happens to unexecutable edges that straddle the boundary of a Subprogram grouping when that grouping is instantiated.

STask Groupings

STask groupings facilitate the representation of IPC of all kinds. They also provide encapsulation for resources to which concurrent access must be controlled. As such, they are used to represent such programming constructs as tasks, monitors, and sequential processes that communicate via messages or remote procedure calls. STasks can also be used to represent semaphore-based concurrency control, albeit somewhat less directly. I defer a discussion of this latter topic, however, until Section 5.2.4.

The design of STask groupings is based on that for Ada tasks, but is not as elaborate. STasks embody the fundamental design features of Ada tasking without any of the bells and whistles. Adornments present in Ada but absent in STasks include delay and terminate options and task creation via calls to new. This last restriction is a consequence of my decision to omit support for data types in SPGs. Programming language constructs that result in the dynamic creation of program entities (e.g., calls to new) typically require the specification of the type of the entity to be dynamically created. Because I have chosen to omit support for data types in my work on SPGs, SPGs provide no mechanism for representing constructs that yield the dynamic creation of program entities.

An STask grouping itself acts as a barrier of abstraction: the contents of an STask are inaccessible to executable portions of the SPG outside the STask. VIDs declared within an STask may not be used outside that STask, RIDs for executable groupings contained within an STask may not be used outside that STask, and no executable edge may cross an STask grouping boundary. STask groupings must have disjoint executable contents, i.e., no executable component of an SPG may be contained inside more than one STask grouping.

Interactions between an STask and the parts of an SPG outside that STask take place through calls to Entry groupings contained inside the STask. Such calls specify the STask containing the entry of interest by referring to the STask's unique TID. This TID, like VIDs and RIDs, is assigned by the SPGM. An STask grouping may contain many entries, so it makes no sense to speak of "invoking" an STask in its entirety. As a result, an STask grouping has no need for executable ports, and it is an error to put such ports on an STask grouping.

STasks are designed to allow SPGs to faithfully and recognizably represent higher-level IPC structures like monitors and tasks. Fundamental to the design of such structures is the assumption that the resources encapsulated inside the IPC structure can be accessed by only a single sequential process. The implication of this assumption for STasks is that

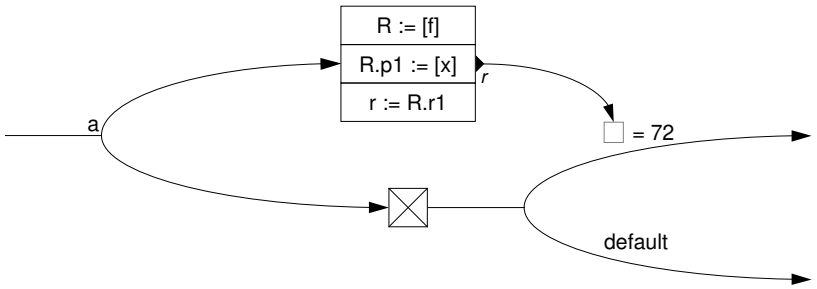


Figure 4-34: An SPG representation of a conditional involving a subprogram call.

there should never be more than a single flow of control within an STask. Given the wide range of software systems that SPGs are designed to represent, however, enforcement of this assumption is neither possible nor desirable.

For example, conditional expressions like "if f(x) = 72 ..." typically result in dual token flows within an SPG, one to represent the flow of control along the edge representing the conditional, the second to enable execution of the function used in the condition. Figure 4-34 shows an SPG representation for this kind of conditional. This construct clearly results in parallel control flow. Parallelism can also arise if a node has more than one executable output port or if an NP_a edge or an NP_i edge (for $i > 1$) is employed. Outlawing such constructs within an STask would pose unreasonable restrictions on the representation of some software systems, especially those based on the dataflow model of computation.

Nevertheless, it is a fact that parallelism within an STask can lead to semantic difficulties. In particular, execution of more than one Accept node at a time violates the fundamental requirement that entry calls within an STask exhibit mutual exclusion. Rather than attempt to outlaw SPG *topologies* that might lead to this kind of behavior, I simply define it to be an error if an Accept node commences *execution* while another Accept node within the same STask is already executing. This has the effect of making the undesirable behavior illegal, while at the same time leaving views free to use whatever SPG constructs they deem most appropriate for subgraphs inside STask groupings.

The restriction that no more than one Accept node within an STask may execute at once also eliminates otherwise troubling race conditions that might arise within an STask. For example, examination of the execution algorithms for Select edges and Accept nodes (Figures 4-28 and 4-18, respectively) reveals that an Accept node is enabled for execution only after execution of a Select edge leading to that node has confirmed that a call is pending for the entry handled by the node. If it were legal for two Accept nodes within the same STask to execute concurrently, it would be possible for the following sequence of events to occur:

1. A single call is made to some entry E within an STask.

2. A Select edge leading to Accept node N_1 that handles E notes that a call is pending for E and passes a token to N_1. This enables execution of N_1.

3. A different Select edge, this one leading to Accept node N_2 (which *also* handles E), notes the pending call and passes a token to N_2. This enables execution of N_2.

4. The nodes N_1 and N_2 both commence execution.

At this point there are two Accept nodes handling the call to E, but there is only one call to handle. The specter of this kind of pathological condition is why simultaneous execution of more than one Accept node in an SPG is disallowed.

Similarly, because an STask is designed to represent IPC structures that encapsulate what is at least conceptually a single sequential process, it makes no sense to manage concurrent access to shared resources *within* an STask. As a result, no STask grouping may be contained within another STask grouping.

In Section 5.2.4 I provide a detailed discussion of IPC representation in SPGs, and Figures 5-5 and 5-6 in that section show how STask groupings, Entry groupings, and Rendezvous groupings work together to emulate the behavior of both tasks in Ada and monitors in Concurrent Pascal.

Entry Groupings

Entry groupings may exist only inside STask groupings. They represent the *interface* through which parts of an SPG outside an STask communicate with that STask. As an interface, an Entry grouping may possess InOut and OutIn ports, which correspond to the tokens consumed and produced when that entry is called, respectively. A token arriving at an InOut port on an Entry grouping is immediately forwarded to the Accept node that is handling the current call to that entry.[8] If there is no multiassignment in the third (parameter values) part of that Accept node waiting for a token from that port, the token is discarded. A token arriving at an OutIn port on an Entry grouping is immediately forwarded to the Call node that invoked the Entry.

Because an Entry grouping is only an interface for IPC, it makes no sense to talk about the *body* of an Entry grouping. Instead, the SPG subgraph to be executed upon receipt of a call to an entry is determined by the Accept node handling the call (see Section 4.3.1). This subgraph is usually contained inside a Rendezvous grouping. It is possible, of course, for a single grouping to be both an Entry grouping and a Rendezvous grouping, in which case the subgraph representing the code to be executed during the rendezvous between caller and callee would be located inside the same grouping that represented the interface to the call. This is a reasonable way of representing monitor procedures, for example, and an example of this approach to monitor implementation can be found in Section 5.2.4.

Rendezvous Groupings

Rendezvous groupings are used to represent the SPG subgraph to be executed during the course of an IPC rendezvous, i.e., during the time when a caller outside an STask is blocked awaiting completion of its call to an entry inside that STask. Rendezvous groupings are similar in both structure and behavior to to Subprogram groupings, but there are important differences between the two kinds of groupings. First, Rendezvous groupings may occur only inside STask groupings, while Subprogram groupings are under no such constraint. Second, Rendezvous groupings are "invoked" through Accept nodes; Subprogram groupings are invoked through Call nodes. Third, unlike Subprogram groupings, no copy of a Rendezvous groupings is made prior to execution of that grouping. That is, a Rendezvous grouping acts

[8] If the token comes from a Parameter node instead of from a Call node, it is possible that the entry invocation for which the token is intended no longer exists. If this is the case, the token is discarded, in accord with the algorithm in Figure 4-14.

not like a template that must be instantiated prior to execution, but is instead executed directly. There is no need to instantiate Rendezvous groupings, because it is impossible to execute more than one Accept node in an STask at once, hence there is never a need to simultaneously execute more than one copy of the SPG subgraph inside a Rendezvous grouping.

InOut ports on Rendezvous groupings are like those on Subprogram groupings in that tokens arriving at such ports are either placed on the source branch originating at the port or, if there is no such source branch, are discarded. OutIn ports on Rendezvous groupings behave slightly differently than their Subprogram grouping counterparts, however. A token arriving at an OutIn port on a Rendezvous grouping is forwarded not to a Call node, but to the fourth (return values) part of the Accept node that invoked the Rendezvous grouping, where it is available for use in multiassignments. If there is no multiassignment in that part of the Accept node that can make use of the token, the token is discarded.

4.4 Annotations

With the exception of tokens, any of the SPG components listed in Table 4-1 may be annotated. Annotations serve three primary purposes in SPGs. First, they provide a mechanism through which standard unary relationships can be expressed. Examples include indicating which nodes are comments, which edges are parts of loops, and which groupings represent scopes. A complete list of standard annotations and their meanings is provided in Tables 4-3 and 5-2, with the former table listing all standard annotations that do *not* relate to loop semantics, and the latter table listing only those annotations that *do* concern loop semantics. I defer an examination of annotations used in conjunction with loops to Section 5.2.1.

I have already introduced many of the annotations in Table 4-3. Start and Stop were mentioned in Section 4.1, and they, along with Return, will be treated again in Section 4.5. Comment, Has Comment, Comments On, and Precedes were examined in Section 4.2, and Section 4.3.3 described Subprogram, Parameter Init, STask, Entry, Rendezvous, Name, and PR Name. The semaphore-related annotations Semaphore and Semaphore Function will be explained in Section 5.2.4. This leaves only the annotations Scope, File, Implicit, Statement, and View-Specific, which I discuss now.

A grouping with a Scope annotation demarcates a region of name visibility for executable components of an SPG. In practice, nodes, edges, and Rendezvous groupings are almost always anonymous, so the SPG entities affected by Scope groupings are variables, subprograms, STasks, and STask entries. Variable names are contained in the Declaration nodes declaring the variables, while names for subprograms, STasks, and entries are represented as Name attributes on the appropriate groupings.

Because of the bewildering array of scope rules employed by languages for software development (see Section 3.5.4), there is little SPGs can do beyond enforcement of the following single rule: each named entity in an SPG must have a unique scope. This scope is defined to be the innermost Scope grouping that contains the named entity, i.e., the innermost Scope grouping containing the Declaration node or executable grouping corresponding to the named entity. It is an error if there is no such scope.

This rule allows for the natural modeling of nested scopes, as shown in Figure 4-35. However, SPG groupings may overlap in an arbitrary fashion, leading to the possibility

Annotation Name	Applies To	Meaning
Name	Any SPG component	The value of the annotation is the user-visible name of this component
PR Name	Executable port on an executable grouping	The value of this annotation indicates the canonical position of this port in a list of parameters or results
Subprogram	Executable grouping	Grouping represents a subprogram
STask	Executable grouping	Grouping represents an STask
Rendezvous	Executable grouping	Grouping represents a rendezvous
Entry	Executable grouping	Grouping represents an STask entry
Semaphore	STask grouping	Grouping represents a semaphore
Semaphore Function	Entry grouping	Value of annotation (either "P" or "V") indicates the role of this entry in the semaphore
File	Grouping	Grouping represents a file; the value of the annotation is the file name
Scope	Grouping	Grouping demarcates a region of name visibility
Parameter Init	Unexecutable grouping	Subgraph inside the grouping initializes variable analogues of formal parameters
Statement	Unexecutable grouping	Subgraph inside the grouping should be presented as a single statement
Start	Executable node not in a Subprogram grouping	SPG execution is initiated by enabling execution of this node
Return	Executable node in an executable grouping	Execution of this node causes execution of the grouping to cease immediately
Stop	Executable node	Execution of this node causes execution of the SPG to cease immediately
Implicit	Any executable SPG component with a Name annotation	Component declaration is implicit
Precedes	Unexecutable edge	Edge sources should precede edge destinations in presentations
Comment	Unexecutable node	Node contains a program comment
Has Comment	Unexecutable edge	Destination nodes contain program commentary about the source components
Comments On	Unexecutable edge	Source nodes contain program commentary about the destination components
View-Specific	Annotation	Annotations of this annotation apply to the SPG component with this annotation, but only for the view named as the value of this annotation

Table 4-3: Standard annotations other than for loops.

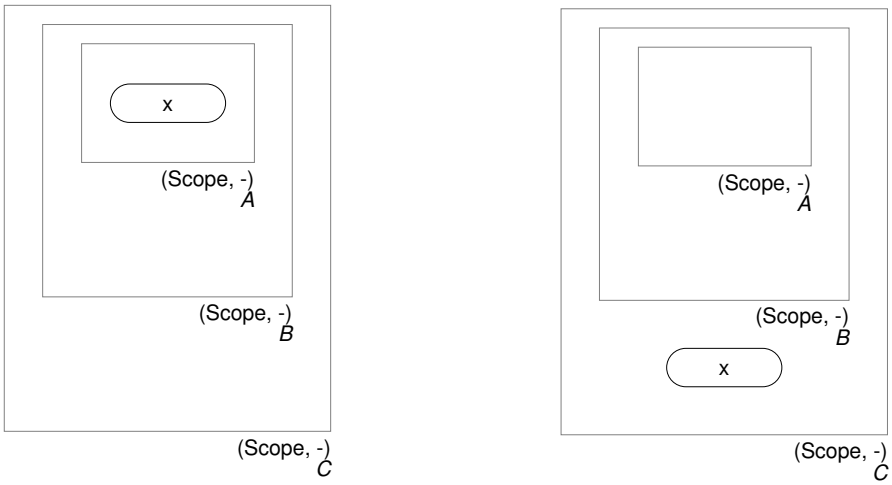


Figure 4-35: Nested scopes. Scope grouping A is contained within Scope grouping B, and B is itself contained in Scope grouping C. At left, the scope of variable x is scope A; x is not available for use in groupings B and C. At right, the scope of x is grouping C, but it is also available for use inside groupings A and B.

that a named entity has no unique scope. For example, consider Figure 4-36, which shows how unnested scopes may overlap to yield both valid and invalid SPGs, depending on whether a named entity occurs within the overlap or without. Of course, a named entity that is not inside any Scope grouping at all also fails to have a unique scope.

A File annotation may be attached to a grouping to indicate that the SPG subgraph inside the grouping represents the contents of a single file in a concrete presentation of the SPG. The value of the annotation is the name of the file. The format of this file name and whether it is absolute (e.g., a full pathname) or relative is left to the discretion of the view creating the annotation. Given that file names are rarely portable across file systems, anyway, it hardly seems worthwhile to try to more formally define how the value of a File annotation should be interpreted.

In many programming languages, declarations of named entities (e.g., variables, subprograms, monitors, etc.) may be either explicit or implicit. This is the case, for example, with real and integer variables in FORTRAN and with functions returning integers in "classic" (i.e., pre-ANSI) C. For views based on such languages, it is useful to be able to distinguish between explicit and implicit declarations of entities within an SPG. This is the purpose of the Implicit annotation. If an executable entity such as a Declaration node, a Subprogram grouping, or an STask grouping possesses an annotation named Implicit, views should, if possible, present the entity as being implicitly declared. If an SPG entity lacks such an annotation, views should assume that it is to be displayed as being explicitly declared.

The Statement annotation is used when a conceptually atomic action in a view must be expressed in an SPG in terms of a subgraph containing more than one node. For example, the BASIC statement,

```
input x, y, z
```

cannot be expressed using a single Multiassignment node, because there is no way to specify that x must be read first, y second, and z third. As a result, the operation must be

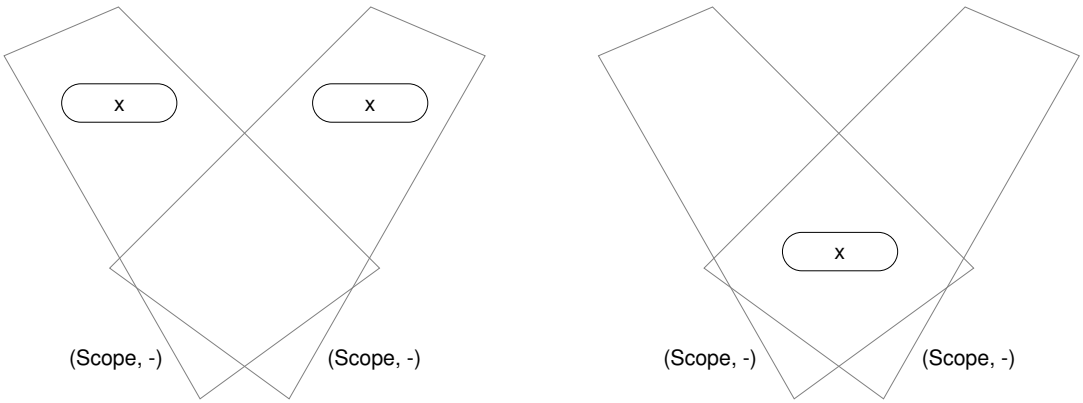


Figure 4-36: Overlapping scopes where each Declaration node has a unique scope (left), and overlapping scopes where the Declaration node has no unique scope (right).

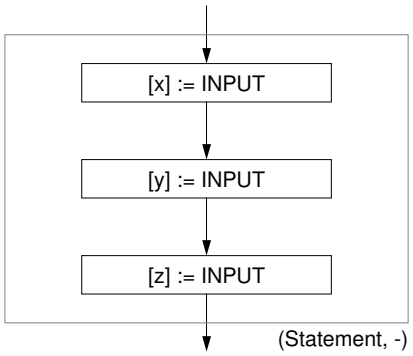


Figure 4-37: Use of a Statement annotation.

broken down into three separate multiassignments, with control edges enforcing the correct sequencing. This faithfully reflects the appropriate semantics, but a straightforward back-mapping from this SPG to BASIC results in the following program fragment, which is unlikely to be acceptable to the developer who wrote the original code:

```
input x
input y
input z
```

Enclosing the three Multiassignment nodes in an unexecutable grouping adorned with the Statement annotation is a solution to this problem, as shown in Figure 4-37. This annotation indicates that the subgraph within the grouping should be displayed as a single statement when views generate presentations.[9]

[9]In the same way that the notion of what it means for one statement to precede another is highly view-dependent, so can we expect the definition of a *statement* to vary from view to view. I do not expect that the view-specific definition of a statement is likely to be the same as the the language definition of a statement for a view displaying that language. For example, I would not expect to see the SPG subgraph corresponding to a C switch statement enclosed in a Statement grouping.

A second purpose for annotations is to allow views to associate view-specific information with the components of an SPG in a standard manner. For example, views might wish to drop markers at certain locations in the SPG, or they might want to store information about the concrete presentation of the view (e.g., line indentations for textual views, object colors and locations for graphical views). In an environment with many different views, each of which is attempting to add its own annotations to a single global name space, the threat of annotation anarchy is far from insignificant. However, the fact that annotations can themselves be annotated allows for the annotation namespace to be partitioned in a controlled manner.

By convention, view-specific annotations of an SPG component are attached to an annotation of that component that is named View-Specific. The value of this annotation is the name of the view to which the annotations are pertinent. Hence, all the view-specific annotations for view V pertaining to SPG component C will be attached to C's annotation, (View-Specific, V). The annotation View-Specific thus acts like a subdirectory within C's global annotation namespace: annotations attached to (View-Specific, V) need not worry about name clashes with annotations for other views, because annotations for any other view V′ will be attached to the annotation (View-Specific, V′) for C, not to C itself. It is an error for an SPG component to have two annotations named View-Specific with the same value. In general, however, it is not illegal for an SPG to contain two annotations with the same name, or even with the same name and value.

The final purpose for annotations is to allow for a restricted form of evolution in the semantics of SPGs. In Section 1.2 I referred to the fact that software development environments must of necessity evolve over time, and in Section 2.1 I explained that this was an aspect of MVDE design that I would not generally address in this thesis. Nevertheless, it is a fact that development environments must change over time, so on purely pragmatic grounds it is essential that a CR for an MVDE provide *some* mechanism for gracefully extending its semantics, even if in doing so the overall integration of the system declines. In fact, a decrease in system integration is all but unavoidable during environment evolution [130], but that cannot obviate the need for a way to evolve (primarily extend) the semantics of a system.

As described in this thesis, SPGs include a fairly small number of standard annotations, i.e., annotations with predefined semantics. I made no attempt to ensure that these would suffice for all possible views that fit the model of Section 3.3. Instead, I focused on providing a framework for the use of annotations to represent unary relationships and on identifying the fundamental unary relationships that must be expressible in an MVDE. Any attempt to do substantially better is almost certain to fail.

As an analogy, consider the command line interface of the X window system. The X Toolkit supports a few basic command line options, including specification of the display to use, the location and size of a program's window, the colors to use for text and for window borders, etc. Because most X clients use the Toolkit, most programs under X automatically offer these command line options to users. This is the well-integrated part of the system. But X clients are free to define their own command line options, so there is no guarantee that different programs won't offer different options, that different programs won't use different option names to achieve the same result, or that different options with the same name won't mean different things to different programs. One result of this *laissez-faire* approach is that the overall integration of an X environment tends to decline as new tools are added to the

system; there are eventually just too many tools using too many inconsistent sets of options. Yet the flexibility offered by the system allows a wide variety of applications to use it.

At some point, however, the designers of X might decide to survey the X clients and their options to identify the ones commonly employed. Having done that, they could then add support for these common options to the Toolkit. Bringing clients into conformance with the new Toolkit might require that many programs be modified for use with the new set of canonical options, but after the changes had been made, the overall integration level of the environment would be higher than it was before the overhaul. The important point to recognize is that it would not have been possible for the designers of X to simply support all the "right" options at the outset, because only by gaining experience with X and its clients would they be in a position to judge which options were "right."

This analysis of the adversarial relationship between integration and evolution is drawn from personal experience and anecdotal evidence. However, it is also in agreement with the rigorous empirical examination of system evolution undertaken by Belady and Lehman [19]. Their first two laws of Program Evolution Dynamics – *the law of continuing change* and *the law of increasing entropy* – neatly summarize the common informal observation that systems tend to change, and as they change, they tend to become less integrated.

Fully integrated environment evolution, then, calls for clairvoyance on the part of SDE designers. Clairvoyance being in short supply, support for environment evolution was not a primary concern of mine in my work on SPGs. Even so, it is noteworthy that SPGs, like X, have the ability to expand their set of system features with predefined semantics. Annotations, therefore, provide a flexible, powerful, and convenient mechanism through which to bring about evolution in an SPG-based development environment.

4.5 Execution

Because an SPG provides a mechanism for representing both executable and unexecutable information about a software system, an SPG need not be executable. For example, an SPG consisting only of unexecutable components is not executable. For an SPG to be executable, it must contain at least one executable node with an annotation named Start. Such nodes are known as *start nodes*. Start nodes are not allowed to be contained inside Subprogram groupings, because it makes no sense to execute nodes inside a subprogram until the subprogram has been instantiated. Similarly, an STask grouping may not contain more than one start node, because an STask is designed to encapsulate a single serial process.

Execution of an SPG is initiated by the SPGM placing tokens on each of the input ports of each of the start nodes in the SPG. Programs that execute without arguments from their external environment can be modeled by giving start nodes a single control input port, while programs expecting data from the external environment (e.g., command-line arguments) can be modeled by putting data input ports on the start nodes. The mechanism by which the SPGM obtains tokens from the external environment for placement on the input ports of start nodes is undefined. Sequential software systems typically have only one start node, but parallel systems may well have many start nodes.

The rules for executing SPGs are similar to those for FSAs and Petri Nets. In general, tokens are created at nodes during execution and are placed on their output ports. The tokens then flow along edges to other nodes and/or edges. A node is eligible for execution

whenever it has tokens on each of its input ports, an edge is eligible whenever a token sits at its destination junction and each of its branch conditions can be evaluated. During execution of an SPG, many tokens typically move around the graph at once, and many nodes and/or edges may be eligible for execution at any given time. This may be the case even for sequential programs, because an SPG representation for conditional constructs may give rise to parallel token flow (see Section 4.3.3 and Figure 4-34).

SPG execution ceases when one of the following conditions occurs:

- A node with an annotation named Stop (a *stop node*) completes execution. If this node has tokens on any of its data output ports, the values of those tokens are communicated to the external environment by the SPGM in a manner that is undefined. This makes it possible for software systems to pass information (e.g., return codes) to the invoking environment. Execution of a stop node is an explicit termination of SPG execution.

- No nodes are executing and there are no tokens remaining in the graph. Under these conditions, it is pointless to continue, because execution is driven by the movement of tokens, and only node execution can produce new tokens. This is implicit termination of SPG execution, akin to running off the end of the main routine in a C program.

As usual, it is possible for a computation to appear to cease even though the underlying software system is still running. This would be the case for a deadlocked system, for example.

The execution algorithm for SPGs is shown in Figure 4-38. After enabling execution of the start nodes in the graph, the SPGM repeatedly, concurrently, and asynchronously performs the following three functions:

- It executes nodes that are eligible for execution. Any number of nodes may be executed simultaneously, provided they all have tokens on each of their input ports.

- It moves tokens from output ports along the source branches that connect to those ports. A token may be moved from such a port as soon as it is placed there, in contrast to the situation with tokens on node input ports, which cannot be removed until each input port on the node has a token. Any number of tokens may simultaneously be in the process of moving from output ports to destination junctions.

- It executes edges that are eligible for execution. Any number of edges may be executed simultaneously, provided they all have tokens at their destination junctions and each of their branch conditions can be evaluated.

Subprogram groupings are instantiated as a result of Call node execution (see Section 4.3.1). This instantiation is accomplished by the SPGM making a copy[10] of the components of the Subprogram grouping, including the grouping itself. The SPGM then places tokens on the copy's InOut ports as specified by the multiassignment making up the second part of the body of the Call node.

A subprogram instantiation completes execution when a node within the instantiation and possessing an annotation named Return (a *return node*) completes execution. One

[10]This is an operational semantics, not an implementation directive. Techniques exist to avoid actually making a copy while still obeying the required semantics [9]. In this thesis, however, I discuss grouping instantiations as if they were always implemented by making a physical copy of an SPG subgraph.

```
(Put tokens on start nodes)
For each node in the SPG with an annotation named "Start":
    For each control input port on the node:
        Create a new control token and place it on the port.
    For each data input port on the node:
        Create a new data token and place it on the port.

(Execute the SPG)
Repeat:
    Perform the following concurrently and asynchronously:
        (Execute eligible nodes)
        For each node that is eligible for execution:
            Execute the node (see Figure 4-39).

        (Move tokens from output ports to destination junctions)
        For each executable output port in the SPG with a token on it:
            If no source branch is connected to the port
                Discard the token.
            Else
                If a token already exists at the destination junction of the edge containing
                the branch
                    Signal an error.
                Else
                    Move the token from the port to the destination junction of the edge
                    containing the branch.

        (Execute eligible edges)
        For each edge that is eligible for execution:
            Execute the edge (see Figures 4-26 and 4-28).

Until (execution is halted via a stop node) or
      (no nodes are executing and there are no tokens on the graph).
```

Figure 4-38: Overall execution algorithm for SPGs.

might expect that a subprogram might also implicitly complete execution when, as in the case of SPG execution, the grouping lacks both executing nodes and existing tokens, but the possible existence of Parameter nodes defeats such a strategy. In general, it is possible for a subprogram invocation to receive a new token from outside the grouping at any time after the invocation is created, because a Parameter node may have access to the RIID for that invocation and thus retain the ability to pass tokens to one or more InOut ports. As a result, subprogram termination must be explicit through the execution of a return node.

Given this preliminary introduction to return nodes, I am finally in a position to replace the incomplete pseudocode for node execution that I presented in Section 4.3.1 with the comprehensive node execution algorithm of Figure 4-39. The figure shows that return nodes are meaningful not only within Subprogram groupings, but within all executable groupings. In the case of Entry and Rendezvous groupings, the rationale for this is identical to that for Subprogram groupings: entries, like subprograms, may be invoked both strictly and non-

```
Perform the node's type-specific behavior.

(Check to see if this is a stop node)
If the node has an annotation named "Stop"
    For each data output port on the node that has a token on it:
        Communicate the value of the token to the external environment.
    Immediately halt all execution of the SPG.

(Check to see if this is a return node)
If (the node is inside an executing grouping) and
  (the node has an annotation named "Return")
    If there is a unique innermost executing grouping containing this node
        G ← that grouping.
        Execute the node (see Figure 4-41).
    else
        Signal an error.

(This is a "normal" node)
Else
    (Remove input tokens)
    For each executable input port on the node:
        Remove the token from the port and discard it.

    (Put tokens on unused output ports)
    For each executable output port on the node that did not receive a token during
    performance of the node's type-specific behavior:
        If the port is a control port
            Create a new control token and place it on the port.
        Else
            Create a new data token with the value UNDEFINED and place it on the port.
```

Figure 4-39: Complete execution algorithm for nodes.

strictly. The case for treating STask groupings in the same manner is currently somewhat weaker, as the primary justification is that it yields a consistent treatment of executable groupings. Recall, however, that the prohibition against dynamic instantiation of STask groupings is based on a lack of SPG support for data types. Given the addition of such support, it would become necessary to provide for the explicit termination of STasks, and that provision is already present in the algorithm of Figure 4-39. The algorithm hence anticipates future SPG support for data types (see Section 8.1).

It is possible for a return node to be contained inside more than one executable grouping. For example, two otherwise disjoint subprograms S_1 and S_2 might both contain a common utility routine R. A return node n within R would then also be within S_1 and S_2. When executing n, the SPGM would have to be able to determine which of the three groupings was performing the return. In this case, static analysis similar to that for determining the scope of an SPG entity (see Section 4.4) could determine that the innermost executable grouping containing n was R, but static analysis does not always suffice. If the grouping

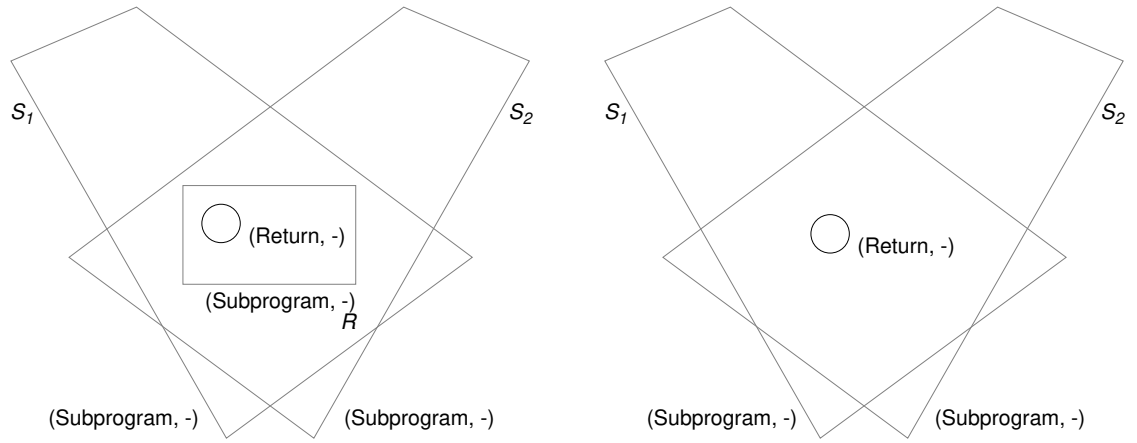


Figure 4-40: A return node contained inside a unique innermost executable grouping (left) and contained inside no unique innermost executable grouping (right).

R were removed from this example, n would still be contained inside both S_1 and S_2, but neither would be nested inside the other. Which grouping should cease execution then? These two scenarios are depicted in Figure 4-40.

The SPGM resolves this kind of ambiguity dynamically, defining the *unique innermost executing grouping* containing the return node as the one that is in fact returning. It is an error if there is no such grouping. In terms of the right-hand diagram in Figure 4-40, if only S_1 or S_2 is executing at the time n is executed, whichever grouping is executing is the one doing the returning. However, if both groupings are executing at the time n is executed, it is an error, because there is no *unique* innermost executing grouping that contains n.

A complete algorithm for execution of a return node is given in Figure 4-41. Although the purpose of a return node is to cause execution of a grouping to cease, the cessation of execution is not quite immediate. Instead, the SPGM halts all node execution and all token flow *except* that of tokens flowing along destination branches toward OutIn ports on the grouping that is executing the return. This ensures that tokens already produced and destined for communication to the invoking node will in fact reach their destination before the grouping invocation ceases to exist.[11] Once a token is placed on an OutIn port, of course, it is immediately transferred to the appropriate invoking node, as described in Section 4.3.3. The SPGM then checks to see if any OutIn ports failed to receive a token during the invocation of the grouping. If so, those ports receive tokens from the SPGM. Control ports receive control tokens, data ports receive data tokens with the value UNDEFINED

Because SPGs support parallelism, it is possible that other nodes in a grouping are executing at the time a return node completes execution. This is problematic when the nodes in question are Accept or Call nodes. If execution of an Accept node is abruptly terminated, the Call node invoking the entry handled at that Accept node may never complete execution, because it may never receive tokens it expects from the entry it called. Similarly, both Call and Accept nodes invoke other executable groupings, so if the execution

[11] For Subprogram and Entry groupings, the invoking node must be a Call node. For Rendezvous groupings, the invoking node must be an Accept node. The issue does not arise for STask groupings, because such groupings are not allowed to have executable ports.

(Stop execution of G)
Immediately halt all execution within G except for the flow of tokens along destination branches to OutIn ports; wait until each such token has reached its port.

(Put tokens on G's unused OutIn ports)
For each OutIn port on G that did not receive a token during execution of G:
    If the port is a control port
        Create a new control token and place it on the port.
    Else
        Create a new data token with the value UNDEFINED and place it on the port.

(If there is a current call to an Accept node in G, terminate it)
If an Accept node in G is currently being executed
    For each target in the fourth (Return Values) part of that Accept node that has not yet received a value:
        If the target is invalid
            Signal an error.
        Else if the target is a data port
            Create a new data token with the value UNDEFINED and place it on the port.
        Else
            Create a new control token and place it on the port.

(Identify all currently executing groupings arising from this execution of G)
GIFH $\leftarrow \emptyset$. *(GIFH = "Groupings Invoked From Here")*
For each currently executing grouping in the transitive closure of the grouping invocations originating from Call and Accept nodes in G:
    Add the grouping to GIFH.

(Halt execution of all the groupings inside GIFH)
For each grouping in GIFH:
    Immediately halt all execution within the grouping.

(Destroy G and the groupings in GIFH)
Add G to GIFH.
For each grouping g $\in$ GIFH:
    Immediately halt all execution within g.
    Remove all tokens inside g or on its ports.
    If g was dynamically instantiated
        Remove the subgraph inside g from the SPG.
        Remove g from the SPG.

Figure 4-41: Execution algorithm for return nodes. The variable G, defined in Figure 4-39, represents the grouping that is returning.

of a Call or Accept node is interrupted, the grouping invocations originating at those nodes no longer make any sense; certainly there is no place to which results produced by those grouping invocations can be returned. Furthermore, such groupings may themselves be in the process of executing Accept and/or Call nodes, so the number of grouping invocations directly or indirectly dependent on an Accept or Call node in a returning grouping is essentially unbounded.

The semantics of return node execution are to immediately halt the execution of all nodes in the returning grouping, including that of Accept and Call nodes. If an Accept node is executing, the SPGM creates tokens to return to the node's caller, with data tokens having the value undefined. If Rendezvous and/or Subprogram groupings are executing at the behest of Accept and/or Call nodes in the returning grouping, the SPGM computes the transitive closure of all currently executing groupings whose invocations originated in the grouping containing the return node, and each of the groupings in that set is destroyed. Finally, the returning grouping itself is destroyed.

The actual steps carried out by the SPGM when destroying a grouping invocation vary slightly, depending on whether the invocation is represented by an *instantiation* of an SPG subgraph (for Subprogram groupings) or by tokens flowing over a static SPG subgraph (for Rendezvous and STask groupings). In both cases, the SPGM removes all tokens from the grouping that is returning. In the case of an instantiation only, the grouping itself is removed from the SPG, as is the subgraph inside it.

I noted in Section 4.3.3 that it is possible for unexecutable edges to cross the boundaries of Subprogram groupings, and now is the time to describe how such edges are treated during instantiation of Subprogram groupings. Given a Subprogram grouping G, an unexecutable edge E, a non-null set of ports P_i in or on G (*inside ports*) to which E is connected, and a non-null set of ports P_o not in G (*outside ports*) to which E is connected, the following occurs when G is instantiated:

- For each output port p_{out} in P_i, a new source branch is created for E. The source branch originates at the copy of p_{out} in the new instantiation of G.
- For each input port p_{in} in P_i, a new destination branch is created for E. The destination branch terminates at the copy of p_{in} in the new instantiation of G.

This process is depicted in Figure 4-42. When a subprogram instantiation completes its execution, the unexecutable source and destination branches that were created to connect to it are also removed.

This behavior for unexecutable edges is founded on the assumption that any unexecutable aspects of an SPG that pertain to the static structure of a subprogram pertain equally well to the dynamic invocations of that subprogram. For example, a comment describing a subprogram is as relevant to an invocation of the subprogram as it is to the source code for the routine. An alternative design would have been to dynamically create unexecutable edges between a subprogram instantiation and the grouping on which that instantiation was based, but in some cases that would make it difficult for views to show dynamic relationships between portions of an SPG. For example, if an unexecutable edge indicates a possible dataflow relationship from a variable definition outside of a subprogram to a variable use inside a subprogram, the dynamic def-use relationship established when the subprogram is instantiated is much more directly expressed by the current behavior of

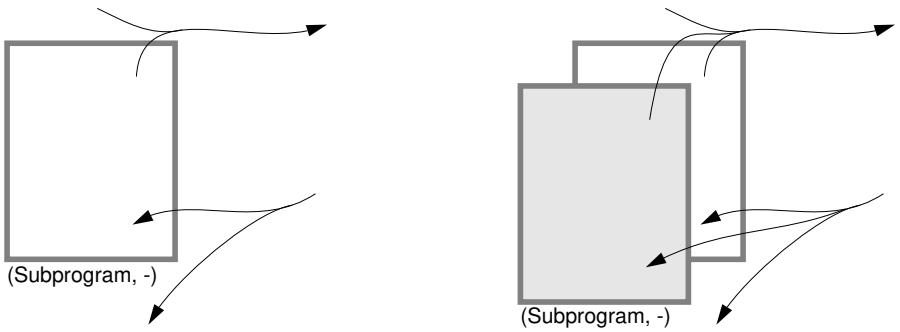


Figure 4-42: Modification of unexecutable edges crossing the boundary of a Subprogram grouping when the subprogram is instantiated. Left: edges and uninstantiated Subprogram grouping. Right: edges and groupings after instantiation.

SPGs than it would be if I had adopted the approach based on linking instantiations to the groupings on which they were based.

The semantics of executing the other kinds of executable groupings (i.e., STask, Entry, and Rendezvous groupings) are somewhat simpler than are those for Subprogram groupings, because the other kinds of groupings require no dynamic instantiation. Like subprograms, however, STask entries may be invoked non-strictly, so it is still necessary for termination of Rendezvous groupings to be explicit through the use of return nodes.

The execution algorithm for SPGs implicitly assumes that there are no existing tokens on the SPG when execution commences. If an SPG was in the process of being executed when the SPGM was asked to begin execution, the SPGM would have to take steps to reset the executable aspects of an SPG to an initial state. Such steps would include removing existing tokens from the graph, flushing pending input and output, and destroying subprogram invocations currently in progress.

4.6 A Simple Example

Figure 4-43 shows an SPG representation for Kernighan's and Ritchie's classic C program for writing "hello, world" to the standard output [108, p. 6]:[12]

```
main()
{
    printf("hello, world\n");
}
```

It should be clear from the SPG that the program in Figure 4-43 contains a single subprogram, main, and that the start node of the program simply invokes main. Both main and the Call node that invoke it are contained in a File grouping (i.e., an unexecutable grouping with a File annotation), and the name of the file is "hello.c". This file is a scope

[12]Because edges can only connect to nodes at ports and because edges are directed and can carry only the type of token matching the ports they connect to, I often omit a graphical depiction of the ports to which edges are attached. For example, the Multiassignment node in the Subprogram grouping of Figure 4-43 must contain a control input port because there is an edge running from the grouping's control InOut port to the node; this port has been omitted from the figure.

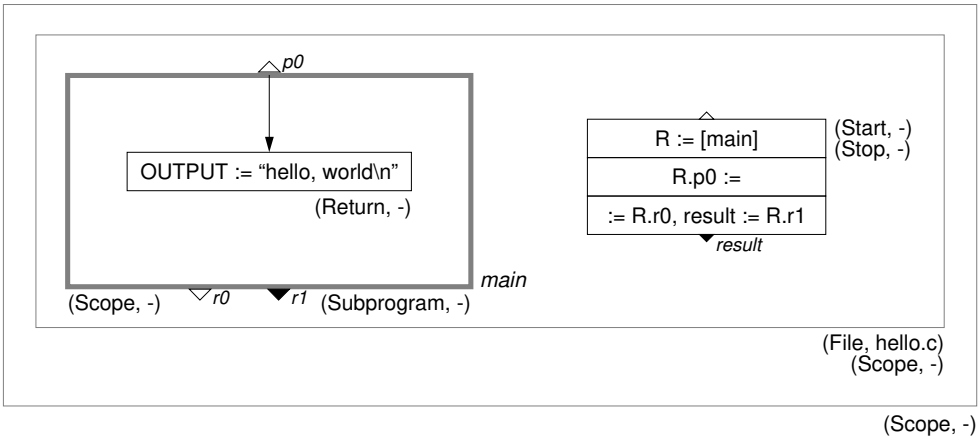


Figure 4-43: An SPG for Kernighan's and Ritchie's "hello, world" program.

in its own right (as is the case for files in C programs), but it is also contained inside a second scope, one that does not correspond to a file. This outer scope represents the global scope of a C program.

Not all languages have a global Scope; Eiffel [127, 128], for example, does not. There is certainly no requirement that an SPG have a global scope. Because the global scope is essentially unused in this particular program, it would have been possible for the view generating the SPG to omit it. To do so, however, would have been to fail to be faithful to the semantics of the C programming language, and one would therefore expect that a C view would be unlikely to take such liberties when generating an SPG for a C program. In addition, omission of a global scope would give rise to ambiguity when other views attempted to make sense of the SPG: should the absence of a global scope be interpreted to mean that there was simply *no need* for such a scope when the SPG was created, or should it be taken to signal a more fundamental restriction — that there should *never be* a global scope? Finding a way to distinguish between these two cases in SPGs is an important topic for future research (c.f. Section 8.1).

There being only one File grouping in this SPG, it is apparent that the source code for the program is entirely contained in a single file called "hello.c". A more representative C program would be composed of multiple files, some consisting primarily of declarations (.h files) and others consisting primarily of definitions. SPGs currently offer no mechanism for distinguishing between the declaration and the definition of variables and functions, so although an SPG can indicate the presence of both .h and .c files for a particular software system, it has no way of indicating that .h files contains only declarations while .c files contain definitions. Adding such support to SPGs would not be difficult, however — see Section 8.1.

The Call node in Figure 4-43 that invokes main is both a start node and a stop node, so execution of this SPG will begin with a call to main and will cease after the call has returned. Execution of this SPG would proceed as follows:

1. The SPGM would identify the Call node as the only start node and would place a control token on its sole (unnamed) input port. This would make the Call node eligible for execution, because it would have a token on each of its input ports.

2. The first part of the Call node would be executed, which would identify main as the routine to invoke. This would cause the SPGM to instantiate a copy of the Subprogram grouping corresponding to main, as described in Section 4.3.3.

3. The second part of the Call node would be executed, which would direct the SPGM to place a (control) token on the InOut port with PR name "p0" of the executable grouping that was instantiated in Step 2. This token would immediately be transmitted along the NP_1 edge (with its destination branch controlled by the implicit condition that always evaluates to true) to the control input port of the Multiassignment node. At this point the Multiassignment node would be eligible for execution.

   The third part of the Call node would also be eligible for execution at this point (because the actions called for by the second part had been completed), but it would immediately block awaiting tokens to be placed on the instantiated grouping's OutIn ports with PR names "r0" and "r1". Until these tokens were available, it would be impossible to perform the assignments in the third part of the Call node.

4. The Multiassignment node would be executed, which would cause the string "hello, world\n" to be written to the standard output. The Multiassignment node is a return node, so after execution of that node, the SPGM would place a control token on the OutIn port named "r0" and a data token with the value UNDEFINED on the OutIn port named "r1". These tokens would immediately be transmitted to the third part of the Call node, which would then be free to complete its own execution by assigning the value of the data token to the Call node's output port called "result". After placing this token on the grouping's OutIn port, the SPGM would destroy the instantiation of the subprogram grouping that had been brought about by execution of the first part of the Call node.

5. After the Call node finished execution, the SPGM would note that it was a stop node. It would then examine the data output ports of the Call node to see if any ports had tokens on them. Finding one on the port "result," it would communicate the value of this token to the external environment that had caused the SPG to be executed in the first place.

4.7 Summary

A Semantic Program Graph is a hypergraph-based representation for both executable and unexecutable aspects of software systems. In accord with the design criteria for a canonical representation outlined in Section 3.5.1, an SPG consists of a small number of generalized components — there are only about three dozen predefined nodes, edges, groupings, and annotations — that can both represent and distinguish between such fundamental concepts as serial and parallel constructs, data-driven and control-driven execution, strict and non-strict parameter evaluation, and deterministic versus nondeterministic execution.

SPGs do not attempt to model all aspects of software systems. Currently, there is no support for data types or data structures. However, SPGs provide a rich, expressive, yet still comprehensible framework for the representation of software systems, and the principles governing their design allow for the addition of new features without unduly compromising the effectiveness of existing SPG components and the views that use them.

The primary purpose of this chapter was to describe the structure and semantics of SPGs. In the next chapter, I show that SPGs are able to meet the view model criterion of expressiveness I developed in Chapter 3. In Chapter 6, I examine how SPGs can be used to form the core of an MVDE, and in Chapter 7, I perform a paper experiment that yields insight into how well SPGs might work out in practice.

Chapter 5

SPGs and the View Model

In Chapter 3 I developed a model for views of software systems based on the essential characteristics of the sequential control flow paradigm, the dataflow paradigm, and the parallel control flow paradigm. I also described four specific representation challenges that a CR for an MVDE must be able to meet: representing loops, representing subprograms and calls, representing name bindings, and representing interprocess communication. In the present chapter, I demonstrate that SPGs are powerful enough to represent each of the semantic features making up the view model, and I provide detailed SPG solutions to each of the representation challenges.

5.1 Representing Model Features Using SPGs

In the sections that follow, I show how SPGs can be used to represent each feature in the view model for software systems.

5.1.1 The Sequential Control Flow Paradigm

SCF1 Sequential, deterministic control flow constructs. Static control flow is represented as control edges and ports, and dynamic control flow is represented as the movement of control tokens through these edges and ports. Simple statements and variable declarations are represented by single nodes, while more complex statements and variable declarations are represented by SPG subgraphs inside Statement groupings. Statement ordering (including gotos) is represented by sequences of nodes connected by NP_1 edges, each with a single destination branch and a null condition. Conditional control flow is represented via NP_1 edges with more than one destination branch, each branch being under the control of a non-null condition (possibly the special condition default). The representation of iterative loops is based on standard annotations; see Section 5.2.1. Subprograms are represented by Subprogram groupings, and calls to subprograms are represented by Call nodes.

SCF2 Identifiers and scopes. The introduction of a new variable identifier is represented by a Declaration node. Such declarations are assumed to be explicit in a view unless the node contains an annotation named Implicit. Variable names may be overloaded without restriction, because references to variables within an SPG

are in terms of unambiguous VIDs. Local variables declared inside subprograms are instantiated along with subprograms and have a lifetime equal to that of the subprogram invocation.

Scopes are represented by groupings with an annotation named Scope, and no restrictions are placed on the relationship between any pair of scopes. Scopes may thus be disjoint, may nest inside one another, or may partially overlap. Whether a grouping is a scope is independent of its other properties, so there is no requirement that, for example, the declaration of a subprogram also introduces a new scope.

SCF3 Operators and Expressions. Simple assignment and arithmetic operators are supported by Multiassignment nodes (see Figure 4-7). The addition of a more extensive set of operators would be straightforward, but is best deferred until greater support for data types is added to SPGs. The standard relational operators are supported by branch conditions (see Figure 4-23). Expressions may involve constants, variables (via VIDs), and computed data values (via data tokens). The two different forms of parentheses in SPGs allow views with uncommon operator precedence levels to ensure that expressions are evaluated correctly within an SPG without requiring the introduction of spurious parenthesis that are visible to view users. Multiple simultaneous assignments are allowed, and deterministic results are guaranteed for the result of such assignments. Input and output of simple data values is supported, as is output of simple literal strings.

SCF4 Files. Files are represented by unexecutable groupings with an annotation named File; the value of the annotation is the name of the file. It is illegal for any SPG component to be contained inside more than one File grouping, so the contents of files must be disjoint.

SCF5 Relationships between system components. Arbitrary relationships between system components are represented by unexecutable edges connecting the SPG representations of the components in question. Unexecutable edges have no semantics associated with them, and this is in accord with the view model's specification of such relationships.

SCF6 Unexecutable ancillary information. The model specifies that such information is simply text, and such text is represented as the contents of unexecutable nodes.

5.1.2 The Dataflow Paradigm

D1 Data-driven execution. Static data flow is represented as data edges and ports, and dynamic data flow is represented as the movement of data tokens through these edges and ports. The creation of new flows of computation is represented by nodes that produce more tokens than they consume, i.e., when the number of data output ports on a node exceeds the number of data input ports on that node. Opportunistic parallelism in execution is inherent in the execution algorithm for SPGs, which allows for any number of nodes to be executed in parallel, subject only to the limitation that a node must have a full complement of input tokens before it is eligible for execution.

D2 Non-strict subprogram calls. Calls themselves are represented by Call nodes, and parameters passed non-strictly are represented as Parameter nodes that receive an RIID from the Call node they work with.

5.1.3 The Parallel Control Flow Paradigm

PCF1 Dynamic process creation. The creation of new flows of control is represented by $NP_{n>1}$ control edges; the destination branches of such edges are typically controlled by the null condition. Views may represent *conditional* dynamic process creation by placing non-null conditions on the destination branches of NP_n control edges.

PCF2 Interprocess communication. The separate processes participating in IPC are represented by STask groupings, and the interface through which other processes communicate with an STask grouping is represented by the set of Entry groupings contained inside the STask. Actual communication between processes is represented by Call nodes that specify a non-null TID. Mutual exclusion within an STask is represented by the single flow of computation that runs inside the STask, and a willingness and/or expectation on the part of that flow of computation to receive communication from other processes is represented by Accept nodes in the SPG subgraph corresponding to that flow of computation.

PCF3 Nondeterministic execution. Nondeterminism can be represented by both NP edges and Select edges. When a token reaches the destination junction of an NP_n edge, copies of that token flow along n nondeterministically selected branches with conditions that evaluate to true.[1] If $n = 1$, only one of the branches with a true condition is chosen, and a copy of the token flows along that branch only. This is nondeterminism in the style of nondeterministic FSAs. IPC-style nondeterminism — that is, nondeterminism in the style of Ada and CSP — is represented by Select edges leading to Accept nodes. This allows views to represent the nondeterminism inherent in client-server architectures where the server must preserve mutual exclusion to shared resources.

Table 5-1 recapitulates the semantic features making up the model for views of software systems and summarizes the primary SPG features used to represent each feature in the model.

5.2 Solving Representative View Problems

I introduced four representation challenges in Section 3.5, and I sketched how they would be handled using SPGs. In the sections that follow, I revisit those representation problems and provide a detailed SPG solution to each one.

5.2.1 Representing Iterative Loops

Based on my analysis of Section 3.5.2, SPGs explicitly recognize but a single form of generalized loop. The SPG subgraph making up the loop itself is enclosed in an unexecutable

[1] Fewer than n tokens are created if fewer than n branches have conditions that are true.

Tag	Model Feature	Primary SPG Features
SCF1	Control flow constructs	Control ports, tokens, and NP_1 edges; executable nodes and groupings; Statement groupings; loop annotations
SCF2	Identifiers and scopes	Declaration nodes, VIDs, Scope groupings
SCF3	Operators and expressions	Multiassignment nodes, branch conditions
SCF4	Files	File Groupings
SCF5	Component relationships	Unexecutable edges
SCF6	Ancillary information	Unexecutable nodes
D1	Data-driven execution	Data ports, edges, and tokens
D2	Non-strict calls	RIIDs, Parameter nodes
PCF1	Dynamic process creation	$NP_{n>1}$ edges
PCF2	Interprocess communication	Stask and Entry groupings, Call and Accept nodes
PCF3	Nondeterministic execution	NP and Select edges, branch conditions

Table 5-1: Summary of semantic features in the view model and the primary SPG features used to represent them.

grouping with an annotation named Loop, and the semantically significant components of the loop are identified by additional annotations as follows:

- The *loop entrance* is an executable destination branch representing the "normal" entrance to a loop. Well-structured programs contain only single-entrance single-exit loops, and for such loops, the normal entrance is the sole entrance to the loop. Not all loops are well-structured, however, so SPGs must admit of the possibility that a loop contains more than one entrance. For a loop with multiple entrances, a view may indicate that one entrance is the normal one by assigning it an annotation named Loop Entrance. (An example of an "abnormal" entrance might be the target point of a goto into an otherwise well-structured loop.)

- A *loop exit* is an executable edge or destination branch representing a flow of control out of a loop. It is identified by an annotation named Loop Exit. SGP components so annotated are typically destination branches. If a loop is followed by a conditional, a nondeterministic choice, or the spawning of a new flow of control, however, it may be convenient to represent a loop exit as an edge with multiple destination branches, each leading to a branch of the conditional, one of the nondeterministic choices, or the initial behavior of a new flow of control, respectively. Many programming languages allow for unstructured exits from what are otherwise single-entrance, single-exit loops (e.g., C's break statement), so an SPG loop may contain more than one loop exit.

- The *loop return* is an executable branch or edge leading from one or more points in the loop body to the top of the loop. It possesses an annotation named Loop Return and is used to represent the points at which a new iteration is initiated. Even structured loops may return to the top of the loop from multiple locations (via, e.g., the continue statement in C), so it is not uncommon for this edge to have multiple source branches[2] or for more than one destination branch in a loop to be a loop return.

[2]This is an example of how the use of hyperedges in SPGs allows common programming control flow to be more directly modeled than would be the case with simple edges.

Annotation Name	Applies To	Meaning
Loop	Unexecutable grouping	Contents of the grouping comprise a loop
Loop Entrance	Executable destination branch	Branch represents the "normal" entrance to a loop
Loop Exit	Executable edge or destination branch	Edge or branch represents an exit from the loop
Loop Return	Executable edge or destination branch	Edge or branch leads from the loop body to the top of the loop for further iteration
Loop Init	Grouping or executable node	Grouping or node represents the code to initialize execution of a loop
Loop Update	Grouping or executable node	Grouping or node represents the code used to update a loop's state on each iteration
Loop Test	Executable edge	Edge contains the condition used to control "normal" exit from the loop

Table 5-2: Standard annotations for loops. As noted in the text, these annotations may also be attached to unexecutable edges originating at the Loop grouping and terminating at the component of interest.

- Many loops perform a set of actions prior to commencing execution of the loop body. For example, for loops typically initialize one or more iteration variables. Within SPGs, such set-up actions are called the *loop initialization* and are represented as a grouping or an executable node with an annotation named Loop Init.

- The *loop update* is a grouping or executable node with an annotation named Loop Update and representing the code used to update a loop's state on each iteration. For example, for loops typically increment one or more variables at the end of each loop iteration.

- An iterative loop usually (though not always) contains a termination condition that controls "normal" exit from the loop. In SPGs, conditionals are always represented by edges with two destination branches (one for each possible outcome of the condition). The *loop test* is therefore an executable edge containing the condition used to control termination of the loop. This edge possesses an annotation named Loop Test. One of the destination branches of a loop test edge must be a loop exit.

The SPG model of a generalized loop, then, consists of an entrance, some exits, some returns, an initialization, an update, and a test. Each component is optional, and each can be identified by a standard SPG annotation generated by the view creating the loop; these annotations are summarized in Figure 5-2. These same annotations may be attached to unexecutable edges leading from the loop grouping to the components; such annotations make identification of the loop components given the loop grouping particularly simple.[3]

[3]I considered assigning to the SPGM the responsibility for generating the annotated unexecutable edges leading from a loop grouping to the loop's constituent components, but this is not always a straightforward task, especially when a loop component is inside more than one loop grouping.

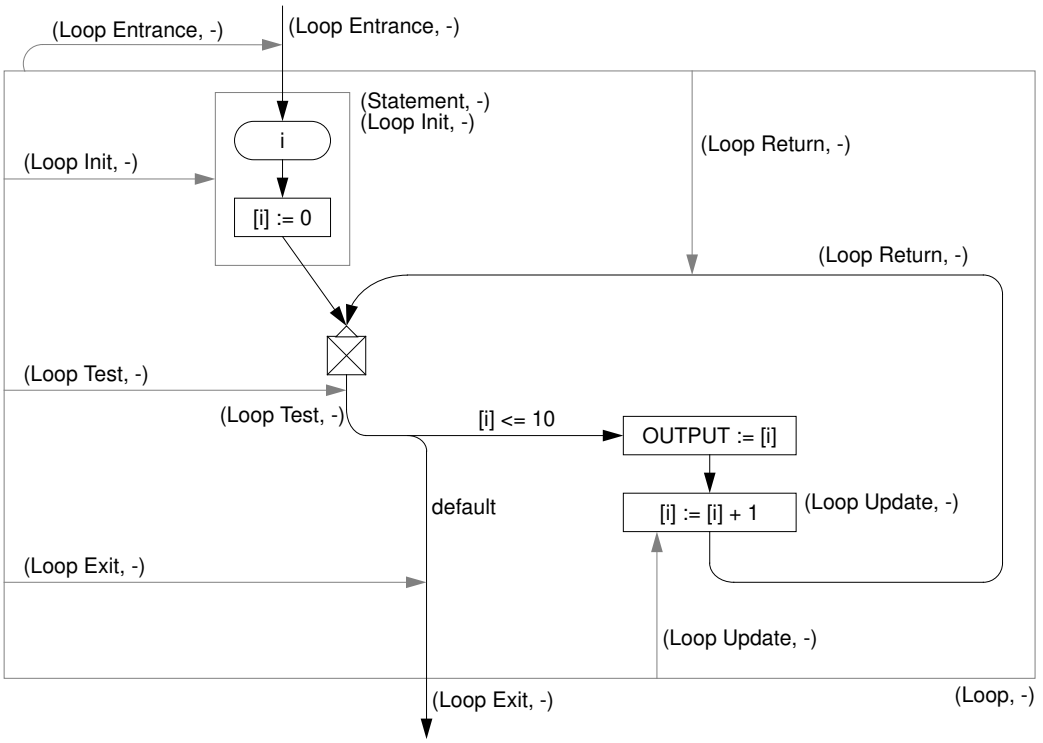


Figure 5-1: An SPG for a loop to print out the numbers 1–10 on the standard output. Executable edges are solid, unexecutable edges are dashed.

Figure 5-1 shows an SPG for the following C++ statement:

```
for (int i = 1; i <= 10; i++) cout << i;
```

A C++-like view can generate a presentation of this loop (and, indeed, of any loop with the recognized components) by applying this template:

Loop Initialization;
`while` (*Loop Test*) {
  Loop Body;
  Loop Update;
}

Application of this template to the SPG in Figure 5-1 yields this code:

```
int i = 0;
while ( i <= 10) {
    cout << i;
    i = i + 1;
}
```

This is semantically equivalent to the original code,[4] but it is not lexically identical, which is what a programmer would expect. For mappings between SPGs and a particular view, however, this problem can be obviated through the use of view-specific annotations. For example, the loop in Figure 5-1 might possess a view-specific annotation indicating that it should be presented as a for loop (as opposed to a while loop or a do loop).

In the more general case — where the view attempting to make sense of an SPG is not the same as the view that generated it — view-specific annotations are unlikely to be present. Indeed, one of my fundamental criteria in designing SPGs was that mappings from SPGs to views should never depend on view-specific annotations. As such, the presentation of an SPG loop is largely left up to the discretion of the view generating the presentation. In many cases this may boil down to a matter of taste on the part of the view writer, especially in C-like languages, where the semantics of for loops are defined in terms of the semantics of while loops [109]. Similarly, views are responsible for translating SPG constructs into view-specific idiomatic constructs in a presentation. For example, it is entirely up to the view doing the mapping whether to display the increment of `i` in Figure 5-1 as "`i=i+1`" or as "`i++`".

It is worth noting that it is not always clear what constitutes a "loop." Consider the following Lisp function for computing a factorial:

```
(defun factorial (n)
  (if (= n 0) 1
    (* n (factorial (- n 1)))))
```

This function contains no loop in the conventional iterative sense, but the conceptual equivalence of tail recursion and iteration is so strong in the Lisp community that many Lisp programmers would claim that "what is going on" is a looping construct and that an SPG for this function should represent it as such.[5] To a programmer seeped in the traditions of Algol-like languages, however, this function contains no loop at all. Rather, its control structure is that of recursion, a strategy utterly distinct from iteration and one which should certainly be represented differently in an SPG.

SPGs cannot arbitrate between such different interpretations, although both can be accurately represented. Instead, decisions such as the precise definition of a loop are relegated to views. In this case, the Lisp view, when translating from Lisp to an SPG, would have to decide whether "what is going on" is iteration or recursion. Once it had made that decision, it would be able to build an appropriate SPG for the function.

In theory, of course, there is no reason why the SPGM couldn't look for instances of tail recursion and replace them with iteration (or vice versa), but in a view-neutral CR like SPGs, it is difficult to justify a choice of one or the other as a canonical form for loops. In fact, the issue of canonicalization of form is a complex matter in its own right — see Section 8.1.

[4] Actually, it is *almost* equivalent. The `while` version employs braces ({ ... }), but the `for` version does not. This means that the while version introduces a scope not present in the for version. This superfluous scope is not present in the SPG, however. It is introduced through application of the template that defines a mapping from an SPG loop to a view-specific loop.

[5] In many Lisp environments, this outlook is based on the fact that tail recursion is automatically converted to iteration by the interpreter and/or compiler prior to execution.

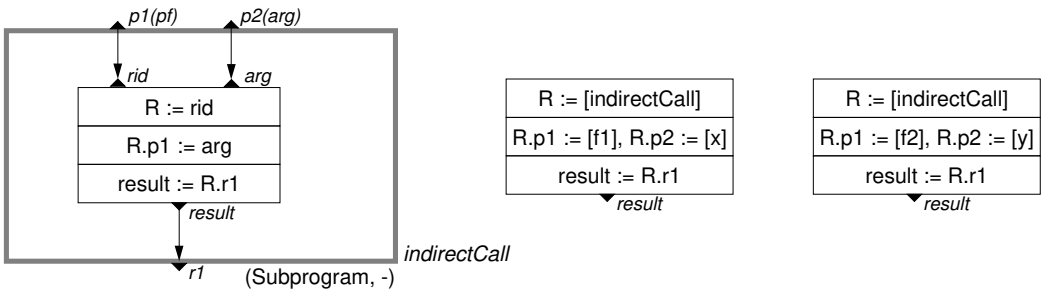


Figure 5-2: SPG subgraphs for the subprogram indirectCall and two different calls to it.

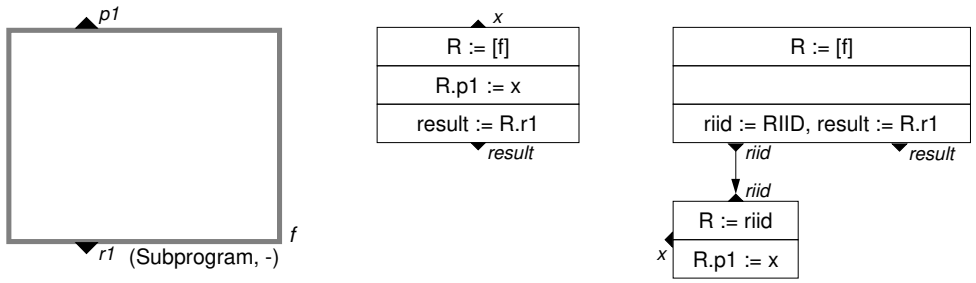


Figure 5-3: SPG representations for a function f and strict and nonstrict calls to it. Contents of the Subprogram grouping are irrelevant and have been omitted.

5.2.2 Representing Subprograms and Calls

In Section 3.5.3, I raised two primary issues relating to the representation of subprograms and calls to them. First, I described the need to be able to represent software systems in which subprograms are themselves passed as arguments to other subprograms, i.e., the need to be able to pass subprograms as parameters. Second, I discussed the importance of being able to pass parameters both strictly and non-strictly to subprogram invocations.

SPGs handle the problem of subprogram parameters by identifying subprograms in terms of a globally unique numerical handle, the RID, and allowing RIDs to be moved through the SPG as values on data tokens. For example, Figure 5-2 shows how SPGs could be used to represent the routine indirectCall introduced in Section 4.3.3. Also shown in the figure are two calls to indirectCall, one to compute the value of f1(x), the second to compute the value of f2(y).

The second problem — that of supporting both strict and non-strict calls — is handled by using Call nodes alone for strict calls and using a combination of Call nodes and Parameter nodes for non-strict calls. In particular, SPGs use a Parameter node for each parameter that is to be passed non-strictly, so a non-strict call to a subprogram taking n parameters will employ one Call node and n Parameter nodes. Figure 5-3 shows how a function f and strict and non-strict calls to it can be represented using SPGs.

There is more to the concept of strictness than I have so far discussed, however, because, given an n-parameter subprogram, one can imagine a gradual decrease in strictness from 0 Parameter nodes (fully strict) to n Parameter nodes (fully non-strict). To the best of my knowledge, every language for software development currently in use employs one of these two extremes, but there is no reason why new views with intermediate semantics might not

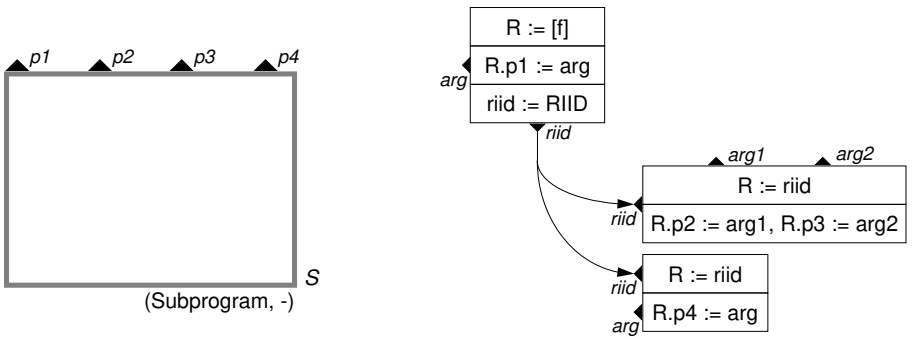


Figure 5-4: A call site passing one parameter fully strictly, one parameter fully non-strictly, and two parameters "partially strictly."

be created.

SPGs naturally offer these intermediate levels of strictness in two ways. First, call sites may pass some parameters strictly (by making assignments to them in the second part of a Call node) and others non-strictly (by passing them via Parameter nodes). Second, SPGs make it possible to constrain sets of parameters such that all members of the set are "equally strict." This capability arises from the fact that a single Parameter node can be used to pass more than one parameter to a routine invocation, and all the parameters so passed are passed at the same time. It is thus possible to use SPGs to represent a subprogram call such that the routine invocation is independent of the availability of its actual parameter values, but the passing of each individual parameter value is constrained by the availability of one or more other parameter values. For example, Figure 5-4 shows a call site to a subprogram S taking four parameters where, S.p1[6] is passed fully strictly, S.p4 is passed fully non-strictly, and S.p2 and S.p3 are passed "partially strictly" — they are constrained to be passed at the same time.

5.2.3 Representing Scopes and Name Bindings

I reasoned in Section 3.5.4 that the proper way to deal with the difficulties associated with binding names to values in a canonical representation for an MVDE was to sidestep it entirely; to insist on the use of unambiguous identifiers within the CR, leaving views to determine how to map unambiguous identifiers back into view-specific name presentations. RIDs and VIDs act as such unambiguous identifiers within SPGs.

5.2.4 Representing Interprocess Communication

I noted in Section 3.5.5 that monitors, CSP-style message-passing, and client/server relationships are all specific instances of a more general concept, that of encapsulated shared resources directly manipulated by a single sequential process and otherwise accessed only indirectly through well-defined procedural entry points into the encapsulation. This general concept is the basis for SPG support for interprocess communication.

[6]The dot notation is used to identify ports on SPG components, so S.p1 refers to the port named p1 on the SPG component named S.

STasks are modeled after Ada tasks, so it is not surprising that the structure of an STask closely mimics that of a task. The primary divergence between the two is that where Ada puts the code for handling an entry call "in line" at the point where the entry is accepted, STasks pull the code to be executed during the rendezvous out into a separate Rendezvous grouping. This can be seen in the SPG in Figure 5-5, which is a representation for the Ada solution to the Bounded Buffer problem that was shown in Figure 4-15. (Appendix A shows a complete Ada program using a bounded buffer and shows how that program can be mapped into and out of SPGs.)

Compared to Ada, STasks are more flexible in their specification of the entry to be handled and the rendezvous code to be executed at a particular accept site. In Ada, both the entry to accept and the code to be executed is statically specified, but in STasks, both of these things may be dynamically specified. As I explained in Section 4.3.1, such dynamic determination of entry and/or rendezvous code is accomplished by storing grouping IDs in variables or passing them as data values onto input ports of Accept nodes.

The mapping between monitors and STasks is somewhat less straightforward than it is for STasks and tasks. There are two primary reason for this. First, while Ada and CSP offer guards to ensure that only messages compatible with the current state are received, monitors have no such provision. (It may in fact be the case that guards were invented for the purpose of addressing this shortcoming of monitors.) This has led to the common idiom in monitor programming of manually coding the "guard" at the top of a monitor procedure, suspending the calling process on a queue if the guard fails. Examples of this convention can be seen in both monitor procedures in Figure 4-16, where the first action in each procedure is to check to see if the variable count has a value compatible with execution of the procedure. If not, the calling process is explicitly suspended.

STasks offer a higher-level abstraction for IPC than do monitors, and as a result there are no SPG operations for manipulating queues of suspended processes; all such manipulations are performed by the SPGM and are hidden from views. When a view translates from a monitor to an SPG, then, it must translate the test at the top of the the monitor procedure into a branch condition controlling token flow into the Accept node(s) handling calls to the STask entry corresponding to the monitor procedure. Examples of this can be seen in Figure 5-6, which shows an SPG for the Concurrent Pascal solution to the Bounded Buffer problem that was shown in Figure 4-16. Mapping from an STask back to a monitor simply reverses this process: branch conditions controlling execution of Accept nodes are translated into tests at the top of the corresponding monitor procedures.

The second difference between the monitor approach to IPC and that employed by STasks is that the duration of a rendezvous with a monitor is always equal to the length of time it takes to execute the monitor procedure that was called. That is, it is not possible for a monitor to do any additional processing for a call after the monitor procedure has returned.

STasks are more flexible than this, as can be seen by comparing Figures 5-5 and 5-6. In the SPG derived from the Ada code, the change in count's value occurs after the rendezvous has terminated, but in the SPG based on Concurrent Pascal, count must be incremented or decremented as part of a monitor procedure, hence during rendezvous with the calling process. When translating from an SPG like that in Figure 5-5 to a monitor view, then, the view performing the translation must examine the graph structure "downstream" from the Accept node in an attempt to determine which instructions must be added to the

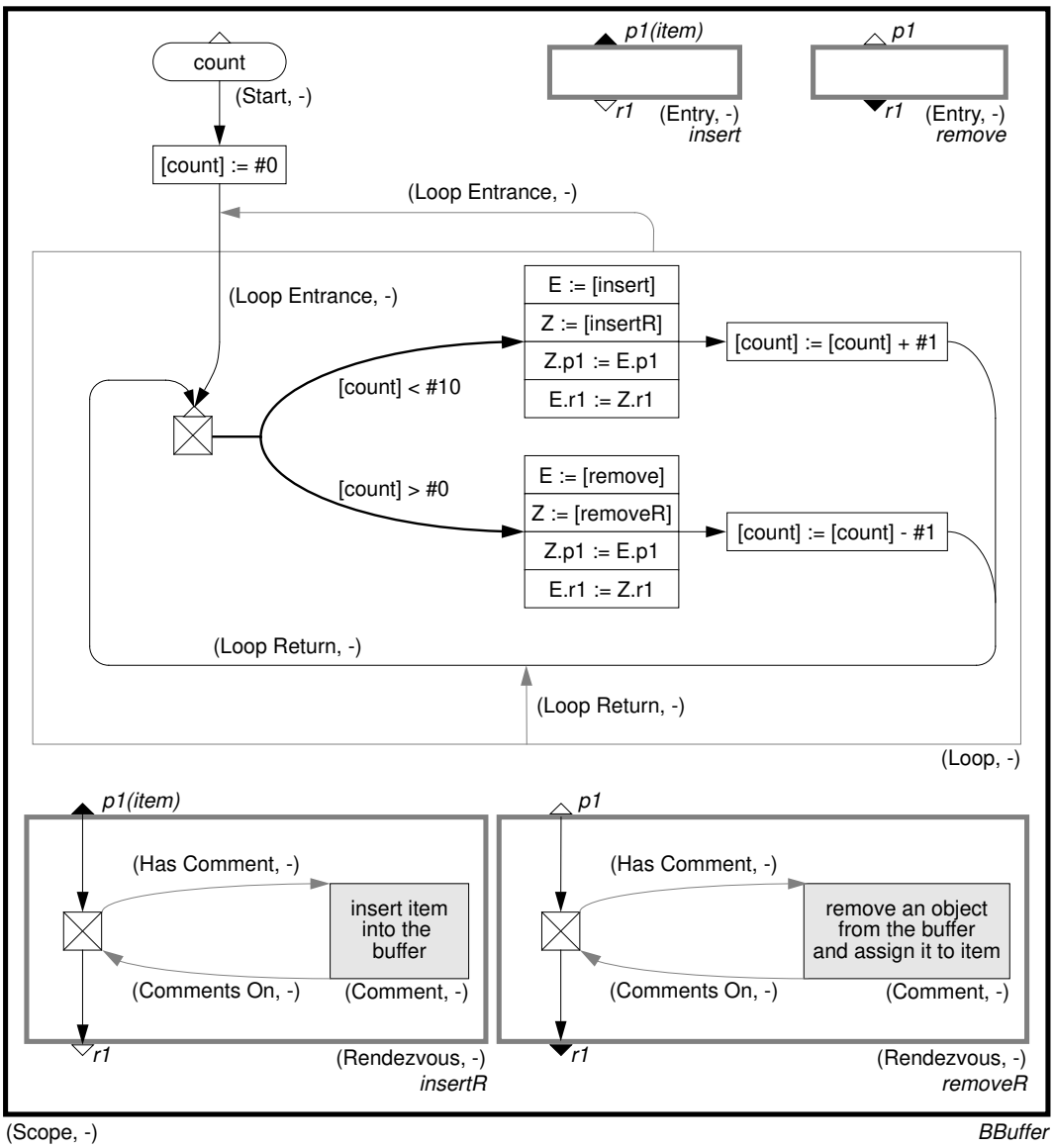


Figure 5-5: An SPG for the Ada solution (Figure 4-15) to the Bounded Buffer problem.

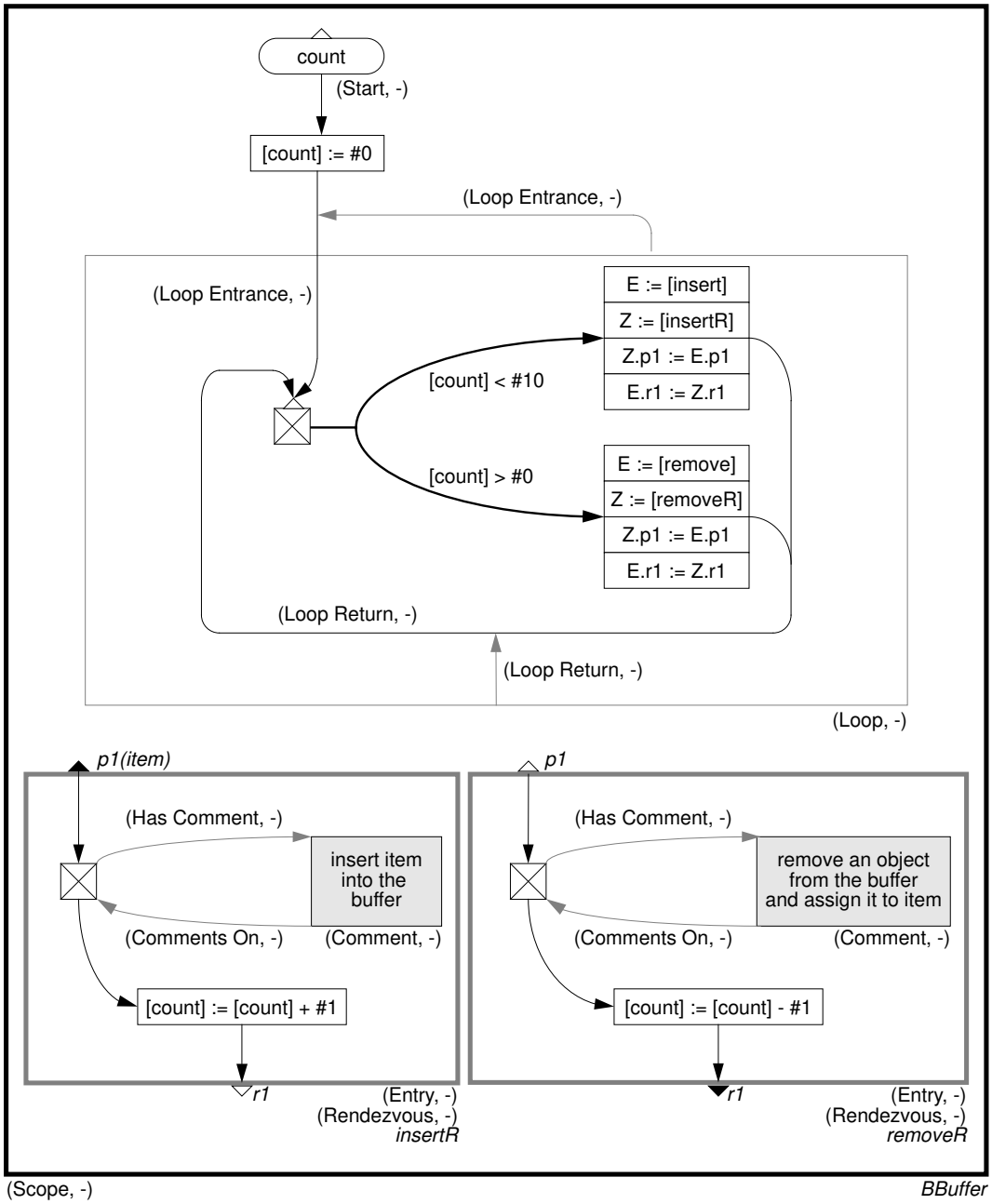


Figure 5-6: An SPG for the Concurrent Pascal solution (Figure 4-16) to the Bounded Buffer problem.

Rendezvous grouping when constructing the presentation of the monitor procedure being called. For the common case of server tasks like the one used in the Bounded Buffer problem, this will be all nodes between the Accept node handling the call and the loop return edge leading back to the top of the loop handling entry calls. Applying this heuristic to the SPG in Figure 5-5 correctly identifies the Multiassignment nodes incrementing or decrementing count as those that must be added to the contents of Rendezvous groupings in order to produce the procedures in the monitor.

Although unidirectional message-passing is easy to represent using STasks and Accept nodes, the particular message semantics of CSP cannot be represented. This is because CSP, unlike STasks, has no anonymous callers: IPC is always accomplished by explicitly naming both the calling and the called process. For a discussion of how this conceptual mismatch might be handled in SPGs, see Section 6.4.2.

The representation of semaphores is problematic because they are at once too ubiquitous to ignore and too low-level to fit into the STask framework I employ in SPGs. Fortunately, the semantics of semaphores can be easily achieved using either monitors or Ada tasks [20], and the mapping between these mechanisms and STasks is, as I have shown, straightforward. Providing the correct execution-time behavior for semaphores, then, is not a challenging problem.

The difficulty arises from the fact that views must be able to recognize an SPG STask emulating a semaphore *as* a semaphore, and this information is not necessarily obvious from the structure of the SPG. One possibility would be to rely on naming conventions for entries of STasks emulating semaphores. For example, an STask with entries named "P" and "V" might be assumed to represent a semaphore. Approaches based on naming conventions are rarely reliable, however. For example, Ben-Ari [20] eschews P and V in favor of "Wait" and "Signal," while Grunwald [75] employs the terms "reserve" and "release."

I therefore employ an approach in SPGs that admits of no ambiguity. An STask representing a semaphore is expected to be assigned an annotation named "Semaphore," and entries within that STask are expected to indicate their function in the semaphore by carrying an annotation named "Semaphore Function" with the value "P" or "V." These annotations make it easy for views to recognize the existence of simulated semaphores in an SPG.

Chapter 6

Using SPGs

In this chapter I consider how SPGs could actually form the core of an MVDE. I focus here on pragmatic issues, such as how views communicate with the SPG Manager (SPGM) and with each other, the functionality of the SPGM, how changes are propagated between views, how views might handle SPG features they can't comprehend, and who is responsible for detecting errors in software systems represented as SPGs. I then examine the kinds of views that are best suited to an SPG-based MVDE. Finally, I describe the prototype SPG implementation I developed, and I summarize the research on object-oriented software development that grew out of my experiences creating the prototype.

6.1 An Architecture for an SPG-Based MVDE

There is more to designing an SPG-centered MVDE than simply turning an olio of views loose on an SPG representing a software system. Such anarchy could lead only to chaos. Instead, some agent must be responsible for ensuring the topological validity of the SPG. Some agent must be responsible for assigning unique IDs to variables and groupings. Some agent must be responsible for interpreting the graph in order to execute the system it represents. In my design for an SPG-based MVDE, the agent performing these functions is the SPG Manager.

The SPGM is the *only* agent that can manipulate an SPG directly. When views want to read from or write to an SPG, they must interact with the SPGM responsible for it. This design treats an SPG as an abstract data type, and the SPGM is the interface through which views interact with the underlying SPG. Such a design not only ensures the integrity of SPGs by restricting access to them to the software making up the SPGM, it also acknowledges that graph structures in general (of which SPGs are but a specific instance) may be implemented using any number of more primitive data structures, and views dependent on the abstract properties of SPGs should be shielded from the implementation decisions made during development of the operations defined by the SPGM.

I explained in Chapter 1 that software developers do not interact with views directly. This is because views are conceptual structures — ways of "making sense" of a software system — and as such may have any of a variety of physical manifestations. For example, a call graph view of a software system is commonly depicted in either a graphical or a tabular format. Each of these formats is a particular *presentation* of the underlying view, and soft-

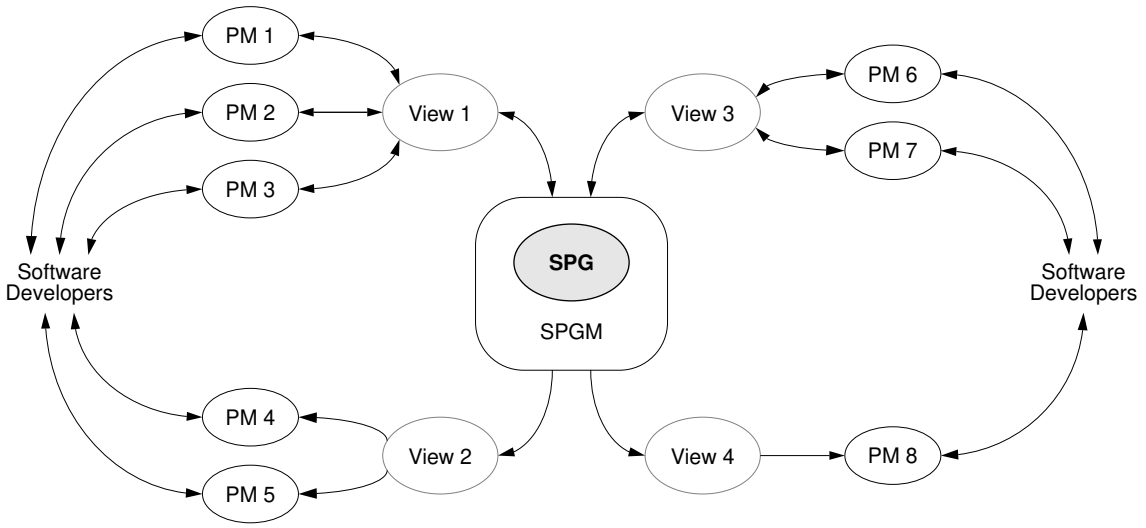


Figure 6-1: The relationship between an SPG, the SPGM, views, PMs, and software developers in an SPG-centered MVDE. Views are drawn with dotted lines to emphasize that they are primarily conceptual entities. Views 1 and 3 are read-write, while views 2 and 4 are read-only.

ware developers interact with presentations, not views. Like SPGs and views, presentations are themselves relatively abstract data structures. Software developers interact with a particular presentation through its user interface, which I call its *Presentation Manager* (PM). This division of an MVDE into separate entities to manage the semantics of the underlying system (the SPGM) and the presentation of those semantics to users (the PMs) has been termed a *presentation-semantic split* [190].

My design for an SPG-based MVDE, then, consists of the following components:

- An encapsulated SPG manipulated by an SPGM.
- One or more views interacting with the SPGM.
- One or more presentations interacting with each view.
- One or more software developers, each of whom interacts with one or more PMs.

This architecture is shown in Figure 6-1, where arrows indicate the direction(s) of information flow. The figure assumes that all presentations are interactive, hence the double-headed arrows between developers and PMs in every case. Views 2 and 4 are designed to be read-only, and for those views information flows exclusively from the SPGM to the views, never in the reverse direction.

Within this architecture, changes to an SPG are brought about by software developers interacting with PMs. When a developer interacts with a presentation, the PM for that presentation initiates the actions that bring about the specified change. There are two types of changes. *Cosmetic* changes may alter the local presentation, but they result in no modification to the underlying SPG. Cosmetic changes generally include mouse and cursor movement, zooming and panning, and simple reformatting of the information in the

presentation. These kinds of changes are semantically neutral, and there is no need to notify the SPGM, much less other views, that they have taken place. Cosmetic changes can therefore be handled entirely within the PM in which they occur.[1]

A change that is not cosmetic is *substantive*: it yields a modification to the underlying SPG. Such changes include adding or deleting a grouping, redefining parts of a node, and renaming a variable. They also include modifications to program comments, assuming such comments are represented using unexecutable SPG components in the customary manner (see Section 4.2).

When a substantive change is made in a presentation, the PM translates the change in the presentation into one or more operations on the SPGM, and the SPGM updates the SPG in accord with these operations. It then notifies each presentation in the environment (including the one that initiated the change) how the SPG was modified, and each PM translates these changes into operations on its presentation. The end result is that substantive changes made to a software system through one PM are uniformly propagated to all other PMs displaying a depiction of that software system.

This approach to change propagation implies that information on a substantive alteration to an SPG presentation is communicated to other presentations as soon as the change is made. Such immediate notification is not always desirable. For example, a developer may wish to engage in a "what if" analysis, whereby the developer can see the effect of a possible change to a system without having to make that (tentative) change apparent to other presentations in the environment. However, immediate notification is not always undesirable, either. For example, a developer adding a new subprogram may want all other presentations to convey the existence of the new functionality as quickly as possible.

This problem — that of determining when to translate changes in a presentation into changes in an SPG — is in essence the issue confronting the database community as it struggles to develop an appropriate transaction model for databases (especially object-oriented databases). This is an active area of research, but no consensus has yet emerged. Explicit support for what-if analysis ("change simulation") is provided by Mercury [104], which implements the feature by modifying a *copy* of the system when a tentative change is made. For my work in this thesis, I assume that immediate notification is always the desired effect, but I discuss the need for additional work in this area in Section 8.2.

6.2 Communication Between PMs and the SPGM

PMs must communicate with the SPGM when they wish to interact with the SPG, and the SPGM must communicate with PMs when the SPG is modified in some way. Because the SPGM and PMs are developed independently (in particular, new PMs may be added at any time), each must have a fixed, well-defined interface that the other can rely on. In

[1] This assumes that no information on things like cursor location, viewport extents, and source code formatting is stored as annotations in the SPG. If this kind of data *is* stored as annotations, changes such as these are considered not cosmetic, but substantive. It is possible to create development environments in which a semantically neutral action in one view can cause a change in other views. For example, selecting a C++ class in the FIELD class browser [166] causes the text editor to automatically jump to the corresponding class declaration in the C++ source code. This kind of information flow between views is interesting and useful, but it is complementary to my research, which focuses on interactions between views as a consequence of modifications to the *semantics* of the underlying software system.

this section, I describe these interfaces, i.e., the SPGM routines that PMs may call and the PM routines that the SPGM may call. The descriptions of these interfaces do not include specific routine signatures (i.e., parameters and/or return types), because, as I explain in Section 6.5, different implementations of an SPGM might adopt different conventions regarding static type-checking in an SPGM or PM interface. The interface descriptions below therefore focus on the information that can be passed across the SPGM interface or the PM interface, rather than on the names and/or arguments of the routines through which that passing takes place.

6.2.1 The SPGM Interface

The SPGM interface offers three broad categories of functionality to PMs. First, it allows PMs to query the structure and content of the SPG, which provides PMs with the ability to traverse the SPG in any way they choose. This capability is essential for PMs being able to map from an SPG into a presentation. Second, it allows PMs to modify the SPG through the addition and/or removal of components of the graph, thus enabling the development of PMs that edit the underlying SPG (both statically and as the SPG executes). Finally, it allows PMs to control the initiation and progression of SPG execution by the SPGM. These three categories of functionality are described in detail in the sections that follow.

Routines That Provide Information About an SPG

- **Identify all the components in the SPG.**
- **Given an SPG component, identify the components it contains.** Examples: given a node, identify the ports in the node; given an edge, identify its source and destination branches; given a grouping, identify the components inside the grouping.
- **Given an SPG component, identify the components containing it.** This is the inverse mapping of the routine above.
- **Given an SPG component, identify its type characteristics.** Examples: given an edge, indicate whether it's an executable edge; given an executable port, indicate whether it's a data port or a control port; given an executable node, identify which type of node it is.
- **Given an SPG component, identify its annotations.**
- **Given an executable grouping, identify its ID.** That is, identify its RID or TID.
- **Given an executable grouping, indicate whether it's a subprogram instantiation.** If it is, identify its RIID and the Call node from which it was invoked.
- **Given a Rendezvous grouping, indicate whether it is executing.** If it is, identify the Accept node from which it was invoked.
- **Given an executable component, indicate whether it is currently being executed.** Whether a grouping is currently executing may not be apparent from a simple inspection of the grouping itself. For example, because SPGs support non-strict parameter passing, it is possible for an Entry or Rendezvous grouping to technically

be executing even if there are no tokens present in the grouping. Information on whether an SPG component is executing is of particular interest to dynamic views, which may need to determine whether edges and/or nodes are in the process of being executed.

- **Given an STask grouping, identify the contents of its queue of pending entry calls.** Each entry in the queue specifies both the Entry grouping being invoked and the node making the call.
- **Given a node, identify its contents.** Examples: given a Declaration node, identify the variables declared there and their VIDs; given a Call node, yield the textual specification of the routine to be called, of how parameter values are to be assigned, and of how return values are to be handled.
- **Given an NP edge, indicate how many branches are to be selected.** That is, given an NP_i edge, identify the value of i. If the edge is an NP_a edge, indicate that in some way.
- **Given a port or branch, identify the components to which it is connected.** Examples: given a port, identify the branches connected to it; given a branch, identify the port to which it connects.
- **Given a destination branch, identify its branch condition.** This will yield the textual specification of the branch condition.
- **Given a token, identify its location.** A token in an SPG may be on a port, on an edge body, or on a branch (as it flows to or from a port). This routine identifies the SPG component on which the specified token currently sits. If the token is not currently on an SPG component, this routine indicates that in some way.
- **Given a port or edge, identify the token that is on the port or edge.** If there is no token present, indicate that in some way.
- **Given a data token, indicate the value it's carrying.**
- **Given a GID, RID, or TID, identify the corresponding grouping.**
- **Given a VID, identify the Declaration node in which the corresponding variable is declared.**
- **Given a VID, indicate the current value of the corresponding variable.**

Many routines in the SPGM yield information in the form of *collections* of entities, e.g., the routine that yields the set of ports on a given node. A collection being an abstraction in its own right, the SPGM must provide a mechanism whereby PMs can iterate through the contents of such collections in a reasonable fashion. There is more than one way to do this. For example, in my prototype implementation (see Section 6.7), any SPGM routine yielding a collection does not actually return the collection itself, but returns instead an iterator object that offers routines for inspecting the contents of the implicit collection on which the iterator operates.

Routines That Modify an SPG

- **Create a new SPG component.** The component may be an executable or unexecutable node, edge, source or destination branch, port, or grouping. It may also be an annotation or a control or data token. The value of a newly created data token is UNDEFINED.

  A newly-created component is not part of an SPG. For it to become part of an SPG, it must be explicitly added (see below). This distinction between creation and addition is important, because some SPG components make little sense in isolation. For example, how should a view make sense of a lone destination branch before that branch has been added to an edge? How should a view depict an input port that has not yet been associated with a node, edge, or grouping? The fact that SPG components are created outside an SPG means that views can build up meaningful semantic structures before adding them to the SPG (at which point other views are notified of their existence).

- **Destroy an existing SPG component.** It is illegal to destroy a component that is still part of an SPG.

- **Add/remove a set of components to/from the SPG.** This routine simply makes the components part of the SPG or removes them from the SPG.

- **Add/remove a component to/from a grouping.**

- **Add/remove an annotation to/from a component.**

- **Add/remove a port to/from a grouping, node, or destination branch.**

- **Add/remove a branch to/from an edge.**

- **Attach/detach a branch to/from a port.**

- **Replace the content of a part of a node.** The parts of a node are its independent textual contents, as described in Section 4.3.1 for each type of node. Declarations and Multiassignment nodes have one part, Parameter nodes have two parts, Call nodes have three parts, etc. Figure 4-19 summarizes the parts of each node type. The content of a node part is a string that is accepted by the grammar corresponding to that node part. Examples: the content of the first (and only) part of a Multiassignment node is a string that specifies the assignments to perform; the content of the first part of a Call node is a string that specifies the grouping to invoke; the content of the third part of an Accept node is a string that specifies what should be done with values returned from the invoked Rendezvous grouping. Fundamentally, this routine replaces one string (the existing content of a node part) with another string (the new content of that node part).

- **Replace a branch condition.** Replace the string specifying a branch condition with a new string.

- **Add/remove a token to/from a port or edge.**

- **Given a data token, specify the value it is carrying.** This value replaces the existing value.

- **Given a VID, set the value of the corresponding variable.** This value replaces the existing value.

Routines That Affect Execution of an SPG

- **Prepare the SPG for a new execution.** Remove all existing tokens from the graph and destroy all existing subprogram instantiations.
- **Initialize the SPG.** Place appropriate tokens on the input ports of each of the start nodes (as per Figure 4-38).
- **Commence unrestricted execution of the SPG.** Begin executing the SPG from its current state (which may not be the same as an initial state) and continue to execute it until the SPG execution algorithm terminates or until the SPGM receives a request to halt execution (see below).
- **Halt execution of the SPG.** If the SPGM is currently executing the SPG, suspend execution at the end of the current single step, i.e., when control reaches the top of the Repeat...Until loop of the SPG execution algorithm. This routine has no effect if the SPGM is not currently executing the SPG.
- **Perform a "single step" of SPG execution.** Formally, a single step is defined to be a single iteration through the Repeat...Until loop of the execution algorithm for SPGs (see Figure 4-38). Informally, it consists of execution of all the nodes that are currently eligible, followed by moving tokens along edges as much as possible.

  This provision for stepping through the execution of an SPG in well-defined increments provides views with a natural basis on which to build views offering controlled execution. Note that for sequential views, a single execution step of an SPG will often correspond to execution of a single statement in the view.

The interface description above specifies *minimal* functionality for an SPGM interface. It does not preclude the inclusion of additional routines for the purpose of improving the convenience or performance of common operations in a PM. For example, if PMs frequently need to produce a list of variables declared in Routine groupings, it is certainly possible for an SPGM to offer a routine that takes as input a Routine grouping and yields as output a collection of Declaration nodes (or a collection of VIDs, or an alphabetically sorted list of variables names, etc.). Clearly, the presence or absence of such routines has no effect on the fundamental ability of PMs to offer views of the SPG to users of the presentations.[2]

6.2.2 The PM Interface

The PM interface exists to meet the needs of the SPGM that arise from its mandate to keep PMs apprised of semantically meaningful changes to the SPG. All such changes must

[2] As an example, in my prototype implementation of SPGs (see Section 6.7), I have found it particularly convenient for the SPGM to offer a routine that, given an SPG component, identifies the groupings immediately containing it. That is, given a component c, it identifies the set of groupings G such that every grouping in G contains c and no grouping in G contains a grouping which itself contains c.

originate with PMs (through calls to the SPGM interface), so it should be of little surprise that the PM interface is in many ways similar to the SPGM interface.

Calls from the SPGM to the PMs must be synchronous, because a PM might want to inspect the SPG as part of its reaction to a call. If PM calls were made asynchronously, the state of the SPG might change between the time the SPGM called the PM and the time the PM inspected the SPG.

The functionality of the PM interface falls into two general classes. First, it allows the SPGM to notify PMs of modifications to the static structure of the SPG. Second, it gives the SPGM a way to inform PMs of changes to the execution status of the SPG. Both of these classes of functionality are described in the sections below.

Routines That React to SPG Modifications

These routines parallel their SPGM counterparts, but when notifying a PM of a modification, it is important that the SPGM make the notification in such a way that the PM is able to discover the context of the change. In particular, it does little good to inform a PM that an SPG component has been removed from the SPG *after* the removal has been performed, because by that time there is no reasonable way for the PM to discover precisely which part of the SPG is affected by the change. Similarly, it is not useful for a PM to know that a component is about to be added to the SPG, because there is no way for the PM to know where in the SPG it will go. As a result, PMs are notified of component additions immediately *after* the addition takes place, and they are notified of component removals immediately *prior* to the removal.

- **A component has just been added to the SPG.** The component is identified.
- **A component is about to be removed from the SPG.** The component is identified.
- **A component has just been added to a grouping.** The component and the grouping are identified.
- **A component is about to be removed from a grouping.** The component and the grouping are identified.
- **An annotation is about to be added/removed to/from a component.** The annotation and the component are identified.
- **A port is about to be added/removed to/from a grouping, node, or destination branch.** The port and the grouping, node, or destination branch are identified.
- **A branch is about to be added/removed to/from an edge.** The branch and the edge are identified.
- **A branch is about to be attached/detached to/from a port.** The branch and the port are identified.
- **The content of a part of a node has just been replaced.** The node, the part, and the old content are identified.

- **A branch condition has just been replaced.** The branch and the old condition are identified.
- **The value of a data token has just been modified.** The token and the old value are identified.
- **The value of a variable has just been explicitly modified.** The VID and the old value are identified. This routine is invoked only when a PM explicitly changes the value of a variable (by calling the appropriate SPGM routine), not when a variable receives a new value during the course of SPG execution. If a PM is interested in tracing changes in variable values, it must do that itself by monitoring execution of nodes that could change the value of the variables of interest.

One of the SPGM routines that modifies an SPG — the one allowing a token to be placed on a port or edge — has no PM counterpart in the foregoing list. This is because the routine allowing a PM to be informed of the arrival of a token at a port or edge is grouped below with the other PM routines that are called during SPG execution.

Routines That React to SPG Execution

These routines are invoked by the SPGM when it executes the SPG and when it prepares the SPG for execution. In general, there are no PM interface routines corresponding to execution-related routines in the the SPGM interface, because an invocation of one of these routines is not, in and of itself, semantically noteworthy. The *side effects* of the execution of such a routine, however, may well be noteworthy. For example, a call to the SPGM routine to initiate SPG execution has, by itself, no effect on the SPG, so there is no PM interface routine indicating that SPG execution has been initiated. However, a side effect of calling that routine is that tokens are placed on the input ports of the graph's start nodes, and the appearance of a token on a port *is* a semantically meaningful event. Thus, the placement of tokens on the ports of the start nodes causes the SPGM to invoke the appropriate PM interface routines.

- **A token has just been added/removed to/from a port or edge.** The token and the port or edge are identified.
- **A node, branch, or edge has just become eligible for execution.** The node, branch, or edge is identified.
- **A node, branch, or edge is about to commence execution.** The node, branch, or edge is identified.
- **A node, branch, or edge has just completed execution.** The node, branch, or edge is identified. If it is an edge, the source and destination branches along which the token flowed are identified.
- **A Select edge is about to commence waiting.** The Entry groupings which, if called or freed for execution, would allow the edge to continue executing are identified. Recall from Figure 4-28 that a Select edge will wait if either (1) the Entry grouping handled by the Accept node the edge's token is destined for is already executing, or

(2) no calls are pending for any of the Entry groupings handled by the Accept nodes to which the edge leads. If (1) is the cause of the wait, this routine will identify the grouping that is currently executing. If (2) is the cause, this routine will identify all the groupings that could be called such that the edge would no longer have to wait. If only a single grouping is identified, a PM can distinguish between the two cases by checking to see if the identified grouping is currently executing.

- **A token is about to flow across a grouping boundary.** The token and the grouping are identified. This routine facilitates the creation of abstract views, i.e., views in which the conceptual states of the computation do not correspond to natural structural features in the SPG. For example, a PM might offer a programmer the ability to select an arbitrary region of a program and have the PM indicate whenever control entered or exited that region. To implement this feature, the PM could create an unexecutable grouping containing the SPG subgraph corresponding to the programmer's region of interest, then monitor token flow across the boundary of this grouping.

- **A subprogram instantiation has just been created.** The instantiation of the invoked Subprogram grouping is identified. This is the *only* PM interface routine called by the SPGM when a subprogram is instantiated. No PM routines are called during construction of the subgraph within the instantiation, nor during addition of the required unexecutable edge branches leading to the instantiation (see Section 4.5).

- **A subprogram instantiation is about to be destroyed.** The instantiation of the invoked Subprogram grouping is identified. This is the *only* PM interface routine called by the SPGM when a subprogram is destroyed. No PM routines are called during destruction of the subgraph within the instantiation, nor during removal of the required unexecutable edge branches leading to the instantiation.

- **The SPGM is about to commence preparation for execution.** When a PM requests that the SPGM prepare the SPG for execution, the SPGM removes all tokens from the graph and destroys all existing subprogram instantiations. Events such as these are normally of interest to PMs, so the SPGM invokes a PM routine each time one occurs. When these events occur as a result of an explicit request to reset the state of the SPG, however, they are typically not meaningful, and most PMs will want to ignore them. This routine in the PM interface allows the SPGM to communicate to PMs that the forthcoming removal of tokens and destruction of subprogram instantiations is due to an explicit request by a PM and is not due to ordinary SPG execution.

- **The SPGM has just completed preparation for execution.** This routine indicates to PMs that the SPG is ready to begin executing normally again and that they should probably resume their usual monitoring of changes to the SPG.

An alternative to the inclusion of the last two routines in the PM interface would have been to specify that the SPGM would call no execution-related PM routines while it prepared the SPG for execution, but it is not inconceivable that some views might wish to

depict the changes occurring in the SPG as it is prepared for execution. Rather than arbitrarily preclude such views from SPG-based MVDEs, I chose to have the SPGM notify the PMs at the beginning and at the end of the preparation process.

6.3 Updating Views

Technically, a view is a set of mappings (see Section 1.1), but conceptually, a view is a way of "making sense" of an SPG. As such, views are entirely virtual. There are typically no data structures in a PM that correspond to a view, although there are likely to be algorithms that implement a view. Different PMs for a single view will usually share the code that allows them to "make sense" of the SPG in the view-specific way.

The existence of an SPG obviates the need for PMs to maintain their own representation of the software system under development, but it is unrealistic to expect an SPG to be the only data structure manipulated by a view. Interactions between developers and PMs imply the existence of a user interface to the presentation, and each PM can therefore be expected to contain presentation-specific data structures that correspond to entities in the user interface of its presentation. For example, a window-based PM would need data structures to keep track of what windows are on the screen, which SPG components are displayed in various windows, etc. Such data structures may be encapsulated in a user interface management system (UIMS) [181, 95].

PMs, then, must directly or indirectly manipulate two sets of data structures: the SPG that represents the software system being presented and presentation-specific data structures that are used to implement an interface to a view. These data structures must be kept consistent with one another, because substantive changes in one usually require changes to the other.[3]

When an SPG is modified, each PM must determine how to update its presentation so as to regain consistency between it and the SPG. There are, in general, two possible ways to approach this task. The "batch[255z" approach is to discard the current presentation, regenerating a new presentation from scratch from the updated SPG. The "incremental" approach, in contrast, is based on limiting and localizing the modifications to the presentation as much as possible.

The great advantage of the batch-style updating of views is its predictability. Suppose $P(G)$ is a presentation of an SPG G, and $U(P, \Delta G)$ is the updated presentation for P given a list of SPG modifications ΔG. If U is a batch-style update function, it is always the case that

$$U(P(G), \Delta G) = P(G + \Delta G).$$

In other words, the presentation resulting from a preexisting presentation and a list of SPG modifications is the same as the initial presentation of the modified SPG. This need not always be the case with incremental update algorithms, where it is possible for an incrementally updated presentation to differ from the presentation yielded by a from-scratch construction.

The downside to batch-style presentation updates is their computational expense, an expense that has motivated much of the research on incremental program analysis algorithms

[3] Not always, however. A substantive change to an SPG need not trigger an update in all views, because some views may not depict the aspect(s) of the SPG that are affected by the change.

for use in interactive development environments [204, 144, 225, 4, 180].

An ideal algorithm for updating a presentation, of course, is as predictable as a batch-mode computation and as efficient as an incremental one. Unfortunately, it may be the case that no such algorithm has been developed for a particular presentation. In cases where no such algorithm is known, there are two ways to approximate this behavior:

- Use an incremental algorithm to update the presentation locally, but add presentation-specific annotations to the SPG to store information about the resulting concrete appearance of the presentation. The data in these annotations can then be used to influence future from-scratch presentations of the SPG so that they appear identical to the result of a series of incremental updates. This is akin to storing view-specific formatting information with the SPG.

- Perform incremental updates by default, but generate a new presentation from scratch whenever the user of a presentation explicitly requests it. This approach has the advantage of allowing the user of the presentation to determine whether the benefit of generating a presentation from scratch is worth the computational (and attendant real-time) cost.

  A second advantage of this strategy is that the requirements for incremental updates can be relaxed. Given that a user can force a from-scratch presentation to be generated at will, implementors of incremental update algorithms can focus on achieving high-performance (i.e., very short response time), possibly at the expense of generality. For example, if an SPG modification cannot be easily mapped into a localized change to a presentation, a PM might simply display the affected portion of the presentation as a "black box" (see Section 6.4.1). Users who found the black box distracting could force a presentation to be generated from scratch, which would (presumably) generate black boxes only when necessary, never merely for convenience.

6.4 Mismatches Between SPGs and Views

Some views map into and out of SPGs more easily than others. For views where one or both of the mappings is difficult, the cause can often be traced to one of the following problems:

- **Missing features in a view.** The view model embodied by SPGs is very rich — much richer than the kinds of view software developers currently employ. What language, after all, supports both strict and non-strict function calls, both serial and parallel computation, both data-driven and control-driven execution? Because views are likely to be semantically impoverished compared to SPGs, it is not uncommon for a view to be faced with an SPG containing features that make no sense for the view. This complicates the matter of mapping from SPGs to views.

- **Differences in semantics for common features.** I took great care in my development of the view model to generalize features whenever possible. This allows many similar view-specific features to be represented in the same way in SPGs. Unfortunately, it is not always possible to be all things to all people, so some generalized SPG features are incompatible with some view-specific expressions of those features. For example, the IPC mechanism supported by SPGs is based on communication from

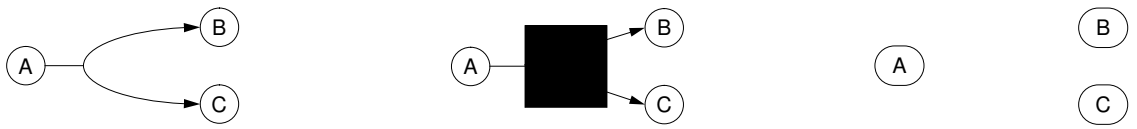


Figure 6-2: SPG containing nondeterministic token flow (left) and two possible ways of representing that in a statechart (middle and right).

anonymous senders, but such anonymity is missing from CSP. This makes it difficult to map CSP into and out of SPGs.

- **Incompatibility between a view and the view model.** My view model is not universal, and some views simply don't mesh well with it. Such views are not expected to be well supported in an SPG-based development environment.

In the sections that follow, I examine these problems in greater detail and I discuss ways in which they might reasonably be handled.

6.4.1 Missing Features in a View

When a feature exists in an SPG, but that features has no counterpart in a view, how should the view handle the feature? There are three general approaches. One possibility is to turn the feature into a *black box* – to have presentations indicate that "something is going on here, but I can't tell you what, because there is no way for me to express it in my language." A second possibility is to *approximate* the semantics of the missing feature with one or more features that do exist in the view. The final alternative is to simply *omit* the feature from the view.

As an example, consider the problem of viewing as a statechart a Petri net containing nondeterminism. Mapping a nondeterministic Petri net construct into an SPG is straightforward, because an NP edge provides the necessary semantics; it also allows for the direct expression of the topological structure in the Petri net. Mapping the resulting SPG into a statechart is problematic, however, because statecharts are designed to specify *deterministic* reactive systems; there is no notion of nondeterminism in the statechart formalism.

A solution to this conundrum is to treat the nondeterministic portion of the SPG as a black box, i.e., to display it in a statechart view in a way that indicates to the viewer that "something is going on here, but I can't tell you what." An alternative is to ignore the construct entirely, omitting from the statechart view any trace of the semantically troublesome portion of the SPG from which the statechart is derived. These two possibilities are shown in Figure 6-2. The black box approach is in general preferable, because it provides more information to the software developer(s) using the view. A black box at least indicates that *something* is happening. Omitting a construct entirely may yield the misleading impression that *nothing* is happening. For example, a developer faced with the right-most depiction of the statechart in Figure 6-2 might well erroneously conclude that there is no relationship between states A, B, and C. The same developer faced with the middle depiction is more likely to conclude that there is *some kind* of a relationship between these states.

Using black boxes to cope with concepts foreign to a view is an effective means of increasing the utility of views in an MVDE and of decreasing the coupling between views,

```
int f(int x) { return 1; }
int g(void)  { return g(); }
main()       { return f(g()); }
```

```
main where
  f(x) = 1;
  g = g;
  main = f(g);
```

Figure 6-3: A C program (left) and a conceptually similar Lucid program (right).

because views need not worry about displaying everything in an SPG; they can simply display black boxes at those points where the SPG makes no sense to them. However, it does make the job of creating a view more complex. Each presentation must augment the concrete syntax for a view (as well as its associated semantics) so that there is some way to physically present the fact that "something unknown is going on." Read-write views must also take care to make the internals of black boxes unmodifiable by users, as it makes no sense to edit something you cannot understand.

A different approach to the problem of mapping from an SPG into a view with a smaller semantic vocabulary is to use approximations in the view for concepts in the SPG that cannot be fully expressed in the view. For example, consider again the two programs in Figure 6-3 that I introduced in Section 3.3.2 as an example of the difference between strict and non-strict program execution. These two programs behave quite differently, but structurally, they are nearly identical, and the only significant difference between an SPG for one and an SPG for the other is likely to be the existence of Parameter nodes in the SPG for the Lucid program and the lack thereof in the SPG for the C program.

Given an SPG corresponding to one of these views, how should the other depict the function calls that are present in the SPG? The calls could be black-boxed, of course, but that would considerably reduce the usefulness of the resulting presentation, because function calls are a fundamental building block for programs in both C and Lucid. Black-boxing all call sites would hide much of "what is going on." A more appealing (and more radical) alternative is to *ignore* the difference between strict and non-strict calls and to simply display SPG call sites as "normal"call sites in the view. That is, the same SPG might be depicted as both the C program and the Lucid program of Figure 6-3, even though the semantics of these programs are not the same. Such an approximation is not true to the semantics of the SPG (and this should certainly be indicated in some manner in the presentation — possibly via comments), but in many cases knowing a partial truth (a function call is taking place) is more useful than knowing nothing at all. As is the case with black boxes, it is probably unwise for views to allow users to modify the internals of approximations that are presented to them.

The need for black boxing and approximation arises from the fact that presentations must be total functions over the domain of SPGs. For features that are present in both the SPG and the view on which the presentation is based, the mapping from SPG to physical manifestation is apparent, but for SPG features with no counterpart in a view, it is not at all obvious how, in general, the mapping should be approached. The easiest solution is to map unknown SPG features into nothing (i.e., to omit such features from the view), but this can be misleading (as in the case of Petri nets and statecharts) or semantically emaciating (as in the case of C and Lucid). Black-boxing and approximation are alternative (frequently

```
BBuffer::
  count, item:  integer;
  count := 0;
  *[
    count < 10; producer?item → count := count + 1 []
    count > 0; consumer?remove() → consumer!item; count := count - 1
   ]
```

Figure 6-4: Solution to the Bounded Buffer problem using CSP. This solution is based on one by Hoare [87, p. 318].

preferable) approaches to the problem of how views can make sense of parts of SPGs that are conceptually alien.

6.4.2 Differences in Semantics for Common Features

In Section 6.4.1, the fundamental stumbling block was in mapping from an SPG into a view, because the SPG had a richer conceptual vocabulary than the view. Mapping from the view into the SPG was not a problem. This is not always the case. It is also possible to have a mismatch between SPGs and a view with respect to a particular feature, even when both the view and SPGs support the feature. An example of this can be seen when trying to represent CSP message-passing in SPGs and, conversely, when trying to express SPG representations for IPC in CSP.

Support for IPC in CSP and in SPGs is similar in many respects. In particular, both are based on the existence of multiple independent processes that communicate through messages.[4] There is a fundamental difference, however, and that is this: in CSP, the recipient of a message must explicitly specify the name of the sender, while in SPGs, the sender is explicitly anonymous. This difference can be seen by comparing SPG representations for a Bounded Buffer (Figures 5-5 and 5-6) with the CSP code for the same problem (Figure 6-4). In the CSP solution, the process BBuffer must explicitly state that it expects to receive messages from processes named producer and consumer. This is fundamentally different from the SPG solution to the problem, where the STask BBuffer specifies only that it expects other (unspecified) processes to call its entries insert and remove.

This kind of mismatch is more serious than one growing out of an inability of a view to express a portion of an SPG, because this kind of mismatch also affects the ability of a view-to-SPG mapping to be written. It is just not possible for an SPG to represent the constraint that, for a given Accept node, the Call node invoking the entry handled by that Accept node must be contained within a particular STask.

This kind of mismatch cannot be handled by black-boxing the construct in question, because in this case the lack of expressive power afflicts the SPG, not the view. Approximation of the construct by some collection of other SPG facilities is in general a possible alternative, but such approximation runs the risk of obscuring "what is going on." This is a serious concern, because, after all, the primary *raison d'être* of an SPG is to accurately

[4] Like Ada and unlike CSP, however, SPGs also allow such processes to communicate through shared memory, i.e. global variables.

and comprehensibly represent the conceptual structure of a software system. If views are forced to resort to approximations in order to produce correct executable behavior in the part of the system they specify, there is a great risk of winning the behavioral battle at a cost of losing the conceptual war. As a result, such approximation should be approached with great caution.

In cases where the semantics of a view construct cannot be more or less directly expressed in an SPG, it must be recognized that the mapping from the view in question to SPGs must of necessity be partial. This being the case, one possibility is to limit the editable scope of the view to those aspects of the system that map well into SPGs. In the case of CSP, this strategy would allow developers to create and modify any aspect of a CSP program *except* those pertaining to message receipt. Carried to the limit, this approach avoids the view-to-SPG mapping entirely by making the view read-only. For CSP, this would mean that developers would be able to look at a software system as a CSP-like program, but they would not be able to modify that system or to create a CSP program within the SPG-based environment.

Even if this read-only approach is adopted, the problem of coping with mismatches in the SPG-to-view mapping must also be addressed. This problem, however, is the same one I discussed in Section 6.4.1, and it admits to the same possible solutions: black boxing, approximation, and simple omission of features in the view.

6.4.3 Incompatibility Between a View and the View Model

In Section 3.3 I developed a model for views of software systems, a model based on the sequential control flow paradigm, the dataflow-based paradigm, and the parallel control-flow paradigm. Not all views fit into this model.

Prolog, for example, is based on an implicit high-level control structure (depth-first search) employing implicit high-level pattern matching (unification) as it manipulates the contents of an implicit global data store. The source code for a Prolog program consists of assertions and rules that are themselves part of the data store, and rules and assertions are added to and removed from the store as the program executes. In some sense, then, Prolog programs are self-modifying: the "program" is the data in the store, and manipulation of this store at runtime yields changes in its contents. This is a demonstrably powerful approach to computation, but it is not one that fits into the model of views that SPGs were designed to support.

As a result, I do not expect that Prolog would be a view well-served by an SPG-based MVDE, and I do not expect that any simple set of extensions to SPGs would materially alter this situation. For views that are as removed from my view model as Prolog is, similar reasoning applies. Fortunately, there are a plethora of useful views that *do* fall within the domain of my view model, and the existence of views outside the model does not diminish the significance of SPGs in facilitating the development of MVDEs.

6.5 Error Detection

Software development is hardly an error-free activity. Syntactic and semantic errors creep in as programs are created or modified, and any number of illegal conditions may arise during execution of a software system. There is no reason to believe that developers using an

SPG-centered environment will suffer from fewer errors than developers using other kinds of environments, but in an SPG-based environment, the problem of error detection takes on a new twist.

In conventional environments, a single agent — typically a language-specific compiler, interpreter, and/or runtime system — is responsible for detecting and reporting errors in the software systems submitted to it. In an environment founded on SPGs, however, PMs and the SPGM must share this duty. This is an outgrowth of the fact that there are two fundamentally different kinds of errors. *View-specific* errors arise when a software system violates a rule that exists in one or more views, but that does not exist within an SPG. For example, creation of a function returning more than one value is illegal in a whole host of languages, but is not illegal in SPGs. *View-invariant* errors, on the other hand, arise when the SPGM is asked to create an SPG structure that is invalid. Both kinds of errors may be static or dynamic. For example, an attempt to nest one function inside another would be a statically detectable view-specific error for a C-like view (but not for an Algol-like view), while having an expression for an RID in a Call node evaluate to an identifier that does not correspond to any executable grouping in the SPG would be a view-invariant dynamic error.

In an MVDE, it often makes little sense to detect view-specific errors at any time other than during *creation* or *modification* of a system's "source code."[5] The alternative — having a presentation issue diagnostics (i.e., error and/or warning messages) whenever the presentation encounters an SPG construct that is illegal in the presentation — is of limited utility in an environment where the portion of the SPG containing the offending construct may have been created by a different view, one in which the construct was legal. As a result, views may well want to assume that everything represented as an SPG has already been statically checked for errors by the view that created that portion of the SPG. An alternative is to use unexecutable groupings to identify SPG subgraphs that are supposed to comply with view-specific constraints. Each view could then ensure that regions of the SPG falling within its realm of constraints always satisfies those constraints.

Even if a view assumes that each part of an SPG was deemed valid by the view that most recently edited it, the view must still deal with the problem of coming up with a presentation for the troublesome portion(s) of the SPG. However, this is precisely the problem I addressed in Section 6.4.1, that of mapping an SPG feature into a view when the view lacks the means to express the semantics of the feature. A viable approach to the problem of view-specific error detection, then, is to assume that "errors" in SPGs actually represent features absent in the view. Such an approach simplifies the process of error detection considerably.

View-invariant error conditions are best handled exclusively by the SPGM. In theory, of course, each view could be made responsible for maintaining the validity of the SPG and for simulating the execution of the software system it represents, but doing so would lead to the kinds of behavioral inconsistencies that a canonical representation was designed to

[5]The term "source code" connotes a single program source — *the* source code — from which all aspects of a system's behavior may be derived. With multiple-view software development, each view is an equally valid source, and the term "source code" no longer identifies anything more authoritative than the appearance of the system in a particular presentation. This shift is similar to that accompanying the move from a document to a *hyper*document. The elements of a document are intended to be read in a particular order — from top to bottom — but the elements of a hyperdocument may be legitimately read in *any* order consistent with the edges in the hypergraph. Note that an SPG is not "source code," because it has no concrete syntax. An SPG is a data structure.

eliminate in the first place (see Section 2.2.5).

Statically, the SPGM is responsible for ensuring that the topological constraints on an SPG are never violated. For example, it must ensure that executable edges terminate only at ports and that Rendezvous groupings exist only inside STasks. There are two fundamental approaches to ensuring that such errors never occur. One approach is to have operations in the SPGM interface fail if they would yield an invalid SPG. Views would then be expected to detect such failed operations and handle them accordingly. Failed operations would never affect the SPG, and there would be no need to inform views in the environment of the invocation of an unsuccessful operation.

An alternative is to use strong typing in the interface specification for the SPGM so that creation of an invalid SPG is simply not possible. For example, the SPGM operation to attach a destination branch to a node could require that the view specify the branch and the *port* to which it should be attached; the port would have to already exist. Similarly, to guarantee that no Rendezvous grouping could be created outside of an STask, the operation for creating a new Rendezvous grouping could require that the view specify the STask inside which it should be created.

This latter approach is the one I adopted in the development of my prototype SPGM implementation (see Section 6.7). The approach works well with strongly typed languages like C++ (which I used for the development of my prototype), because potential errors in SPG construction can be detected at the time the code for the *presentation manager* is compiled. It is less effective when views are developed in weakly typed languages like Lisp.

The SPGM is also responsible for detecting and reporting to views the presence of errors that arise at run time, i.e., while the SPGM is interpreting the SPG. Such errors include references to invalid VIDs, RIDs, TIDs, etc.; the simultaneous presence of more than one token on an edge; and commencing execution of an Accept node while another Accept node in the same STask is already being executed.

The division of labor between PMs and the SPGM in error detection, then, is in some sense the natural one: the SPGM detects and reports (to PMs) conditions that are *always* errors, and the responsibility for detecting and handling conditions that are valid within the SPG but are invalid in one or more views is relegated to the PMs. This is a similar situation to the one I described in Section 3.5.4, where I examined the problem of disambiguating references to identifiers. Just as the bewildering array of view-specific rules for name disambiguation made it impossible for the SPGM to reasonably attack that task, so does the disparate variety of view-specific error conditions require that the onus of error detection, too, falls on views.

6.6 Suitable Views

In Chapter 3 I developed a model for views of software systems. That model grew out of an examination of the characteristic features of three broad paradigms of computing: sequential control-flow, dataflow, and parallel control-flow. This feature-driven approach was appropriate for determining the necessary *expressiveness* of SPGs, but it serves less well for the purpose of characterizing the kinds of views that software developers actually *use*. The variance in the aspects of a software system that are of interest to programmers is considerably smaller than is the variance in the combinations of language features exhibited by the different languages in use. In practical terms, this means that the kinds of views that

	View Class		
View	**Static Execution Flow**	**System Structure**	**Execution**
Source Code	✓		✓
Flowchart	✓		✓
Call Graph	✓	✓	✓
Runtime Stack			✓
Resource Profile			✓
Test Coverage			✓
Build Dependencies		✓	
Petri Net	✓	✓	✓

Table 6-1: Sample views and the types of information they provide.

programmers find useful tend to break down not along the lines of the computing paradigms I described in Chapter 3, but along the following lines instead:

- **Static execution flow:** views that show some aspect of the static paths along which control of execution may flow through the system. For procedural languages, such views tend to show conventional control-flow information; for dataflow languages, they tend to show dataflow information, because the flow of control in a dataflow language follows the flow of data. Examples of views of this ilk include the textual source code of conventional programming languages, call graphs, and statecharts.

- **System structure:** views that abstract away details of a system in order to convey information about the overall system structure. These views are often based on static execution flow. Examples include class hierarchies, call graphs, and Petri nets.

- **Execution:** dynamic views that show the changing states of a system during execution. Such views are often animated versions of static views of system structure, including those listed above.

Table 6-1 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. As the table indicates, many views provide more than one type of information. Abstract views of a system's structure often retain some information on static execution flow; this is the case with call graphs and Petri nets, for example. In addition, a static view can frequently double as the basis for a simple dynamic view, typically through the selective highlighting of the view's components during execution.

In the sections that follow, I consider each of the views listed in the table. For each view, I briefly describe the information it offers to users, and I provide a sketch of how it can be implemented in terms of an SPG.

6.6.1 Source Code

Source code is the primary view through which programmers see the software systems they work on. All other views are of secondary importance. In the words of one researcher in the field of software development environments, "source code is gospel."

The fundamental motivation behind SPGs is the faithful representation of this gospel in an accessible format. It is the underlying theme running through Chapter 3, which explained which semantic concepts are important enough to warrant representation, Chapter 4, which described in detail the structure and semantics of SPGs, and Chapter 5, which showed how SPGs were able to represent the chosen semantic concepts in an unambiguous manner. For an example of how the features of a Pascal-like language map into and out of an SPG, consult Chapter 7 and Appendix B.

6.6.2 Flowchart

A flowchart is a classical diagrammatic representation of the possible paths of control flow through a program. It is designed for use with languages employing the sequential control-flow paradigm.

For SPGs containing only features present in this paradigm, mapping between a flowchart view and an SPG is not a difficult task. Flowchart boxes generally correspond to nodes or to Statement groupings in the SPG, and flowchart arrows correspond to control edges in the SPG. Decision boxes in flowcharts correspond to branch conditions in SPGs. Each Routine grouping in an SPG corresponds to a separate flowchart.

When mapping from an SPG that employs semantics outside that of the control-flow paradigm, a flowchart view would have to black-box or approximate the features missing from the view (see Section 6.4.1).

Flowchart views lend themselves to the presentation of execution information by the dynamic highlighting of the box currently being executed.

6.6.3 Call Graph

A call graph depicts information on which routines in a program may call which other routines. The information is statically determined, so indirect calls (i.e., call sites where the called routine is determined at runtime) are typically omitted from the view.

Implementation of a traditional call graph view for an SPG is straightforward, because routines are easy to identify (they correspond to Routine groupings) and call sites are equally easy to identify (they correspond to Call nodes with empty TID specifications). Building a call graph view atop an SPG allows for the view to provide additional information, however, such as which calls are strict and which are non-strict. (A good heuristic for determining whether a call site is non-strict is to see whether an RIID value is placed on the Call node's output port). It could also be used to show calls between STask groupings, thus yielding a "message graph."

A graphical presentation of a call graph view can be dynamically animated by highlighting the image of the Routine grouping invoked when a Call node is executed and by unhighlighting that image when a Return node inside the grouping is executed.

6.6.4 Runtime Stack

A runtime stack view is used to display which routines are currently executing and to show current values of the variables declared in the executing routines. Such views are traditionally graphical, and there is usually a meaningful relationship between the physical location of the information corresponding to a called routine and that of its caller. Often,

the graphical depiction of the caller's stack frame is directly below (or above, depending on whether stacks are shown growing up or down) that of the callee's stack frame.

Implementation of a runtime stack view is similar to that for a call graph, because making a call causes creation of a new stack frame and returning from a call causes destruction of a stack frame. Locating the variables declared within a stack frame is equivalent to locating all the Declaration nodes within all the Scope groupings that are within the Subprogram grouping instantiation (including the instantiation itself, if it is a scope). Given the VIDs of the variables in the stack frame, the PM may track changes in their values by querying the SPGM for their values after execution of each node that might affect them. A runtime stack view could allow developers to modify the value of variables visible in the view, thus offering some run-time editing capabilities.

The natural appearance and disappearance of stack frames during program execution makes a runtime stack view a natural candidate for a dynamic view. As in the case of a call graph view, different colors or line styles could be used to distinguish between stack frames that are currently executing, those that are waiting for a called routine to return, and, possibly, those that have already returned (thus yielding a historical view of the computation).

A sample implementation of a Runtime Stack view can be found in Section B.5. Observations about this implementation are in Section 7.6.

6.6.5 Resource Profile

Profiling views keep track of resource usage as a function of location in a software system. For example, a profiler might count the number of times a statement or routine is executed; the number of times a particular branch of control is taken; or the elapsed amount of CPU time consumed by a routine.

I noted in Section 3.3 that an SPG is not designed for the support of low-level views such as those based on CPU utilization, but for views based on higher-level abstractions, implementation of the machinery needed to collect the necessary usage data is straightforward. The basic approach is to associate view-specific information with the SPG objects of interest, updating the information as the SPG is executed. The data is stored as the values of view-specific annotations on the relevant SPG objects. For example, to keep track of the number of invocations of a routine, a view could assign a view-specific annotation to the appropriate Routine grouping, initialize its value to 0, and increment it each time the routine is called.

Resource profiles can be overlaid on a static view of a system (e.g., a call graph, etc.) and updated during execution to provide a form of program visualization.

6.6.6 Test Coverage

Test coverage views of a software system provide some form of indication of how much of the system was exercised during the course of a test run. A number of approaches to software testing have been proposed, along with an even greater number of metrics for measuring the effectiveness of the different approaches [18, 175], but in practice, most software developers who formally test their software[6] rely on relatively simple structural testing criteria, such

[6]Those who do not may have good reason. Petschenik [156] has pointed out that for the large system he works with (over 2 million source lines), as much code would be required for the formally required test cases as for the system itself.

as statement or branch coverage.

Implementation of a view based on statement or branch coverage can adopt the strategy sketched above for resource profiling: use view-specific annotations to identify which nodes, Statement groupings, edges, branches, etc. are executed during the course of a program run. The coverage of a test input is then equivalent to the portion of the entities of interest (e.g., nodes, branches, etc.) in the SPG that are so identified.

6.6.7 Build Dependencies

A view of the build dependencies of a software system shows which parts of a system would require recompilation if a modification were made to a given part of that system. The granularity of a "part" varies, but it typically corresponds to the smallest compilable portion of the system, usually a file or routine. Some languages (e.g., Eiffel) have dependency-checking built into the language. In other languages (e.g., C), tools external to the language conservatively estimate dependencies between parts by noting how source files relate to one another. This is, for example, the approach adopted by makedepend.

In principle, a view of build dependencies could perform a complete data flow analysis of the SPG in order to determine the smallest possible set of dependencies between parts, an option made possible by the fact that an SPG represents an entire software system, not just the subsystem contained in, say, a single file (as would be the case with a compiler for C or FORTRAN). Such a view could be expensive to produce and maintain, however, so an approximation might be preferable.

One way to (conservatively) approximate dependencies between parts is to say that a part (i.e., a subgraph) P_1 is dependent on a part P_2 if either of the following conditions hold:

- P_2 contains a grouping invoked from P_1. This would be the case if P_1 contains a Call or Rendezvous node and the corresponding routine or Rendezvous grouping is contained in P_2.
- P_1 uses the VID of a variable declared in or the GID or TID of a grouping contained in P_2.

It is difficult to neither construct nor maintain this kind of approximation, but, because it is based on the *semantics* of the system represented by the SPG, it yields more accurate information than a syntactic makedepend-like approximation.

6.6.8 Petri Net

Petri nets are an FSA-like graphical depiction of possible state changes. They are often used to reason about the possible behavior of asynchronous parallel systems. The implementation of views based on Petri nets is explored in Appendix A.

6.6.9 Conclusion

The views listed in Table 6-1 and discussed in the preceding sections comprise not an exhaustive accounting of the kinds of views that are suitable for use with Semantic Program Graphs, but are instead suggestive of the breadth of views that can be effectively based on

Figure 6-5: Approximation of a hyperedge (left) in the GELO presentation (right). The edge body and each of its branches is represented as a simple edge, and the edge's source and destination junctions are represented as GELO nodes.

SPGs. They demonstrate that views providing information on a system's static execution flow, its system structure, and/or its dynamic execution can all be successfully implemented in terms of SPGs. In addition, views such as runtime stack, test coverage, and build dependencies show that an SPG-based MVDE can offer a broad range of views of software systems, including those that are only tangentially related to the physical "source code" of the system.

6.7 Experience with a Prototype Implementation

I implemented a prototype SPG-based environment for the purpose of experimenting with the ideas described in this thesis. The prototype supports a subset of the SPGM and PM interfaces described in Section 6.2, offering many of the routines that create and execute SPGs, but offering few routines that allow SPGs to be edited. I also implemented three views, each with a single presentation manager.

The first view was originally designed only for debugging. My goal was to simply draw a pictorial representation of an SPG. To produce an image of the graph, I took advantage of the automatic layout capabilities of the GELO library [167], and the resulting view is known as the GELO view. Implementation of the GELO view was substantially less straightforward than a simple data-structure dump, because GELO is unable to draw pictures of hyperedges or of edges that terminate at other edges. It was therefore necessary for the GELO PM to cope with the problem of missing features in a view (see Section 6.4.1).

I was able to overcome GELO's inability to display hyperedges by adopting an approximation strategy. Each hyperedge in an SPG is represented by GELO as a combination of GELO nodes and simple edges. The edge's body and each of its branches is represented by a simple edge in GELO, and GELO nodes are introduced to represent the edge's junctions. This approximation is shown in Figure 6-5.

It would also have been possible to approximate SPG edge branches that terminate at other edges (Figure 6-6 shows one way of doing it), but I chose to use a black-box approach to this mapping problem instead. The current GELO presentation black-boxes edge branches that terminate at other edges (i.e., it fails to display them), but it indicates their presence by drawing the affected edge branch (the branch that is the destination of the branch being omitted) using a different line style than is used for branches with no incoming branches. Hence, "normal" branches are drawn using a solid line, while branches with suppressed input edges are drawn using a dashed line.

Figure 6-7 shows a statechart specification for a simple software system, and Figure 6-8

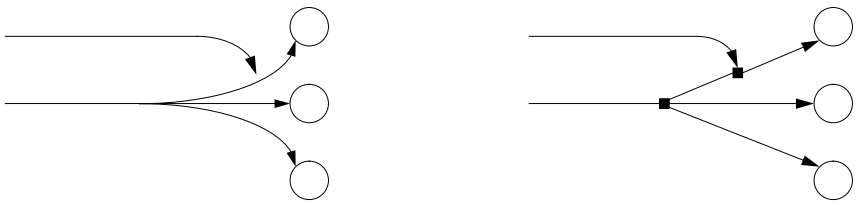


Figure 6-6: Possible approximation of an edge terminating at another edge (left) by GELO (right). The SPG branch containing the input port is represented by a two simple edges and a GELO node.

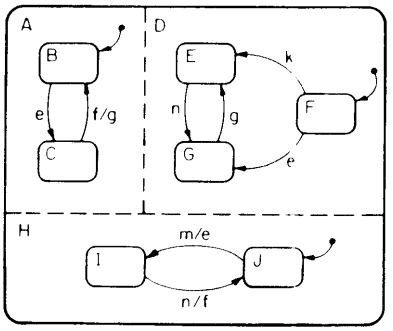


Figure 6-7: A sample statechart. (This is Figure 18 of Harel's *CACM* paper on statecharts [80]. It is copyright © 1988, Association for Computing Machinery (ACM). Used with permission.)

shows the GELO presentation of an SPG for that system.[7] The picture produced by the GELO presentation is anything but beautiful, but it succeeds in its mission to visualize an SPG. In the presentation, SPG nodes are depicted as GELO nodes, SPG edges are depicted as I have just described, and SPG groupings are depicted as GELO tilings, with the grouping name in the upper tile and the contents of the grouping in the lower tile (see Figure 6-9). Annotations are not shown in the graphical depiction of an SPG, but are instead displayed on a per-object basis when a user explicitly asks to see them. Such a request is generated by clicking the mouse on top of a displayed object.

The GELO presentation is interactive in other ways. When a user clicks on an object in the presentation, not only are its annotations displayed, it also becomes the *current object*. A small set of interactive commands applies to the current object. One command highlights all the sources of the current object, meaning all SPG objects containing output ports connected to edges containing destination branches leading to the current object. An analogous command highlights all the sinks of the current object. A third command causes tokens to be placed on all the input ports of the current object that currently lack tokens, thus enabling the current object for execution. Three additional commands control execution of the SPG. The first resets the SPG to an initial state, the second performs a

[7] The mapping from statecharts to SPGs I employed in my prototype was sufficient for the statecharts with which I was experimenting, but it was not a fully general mapping. Such a mapping is possible (excluding, e.g., timing constraints), but the resulting SPG is more complicated.

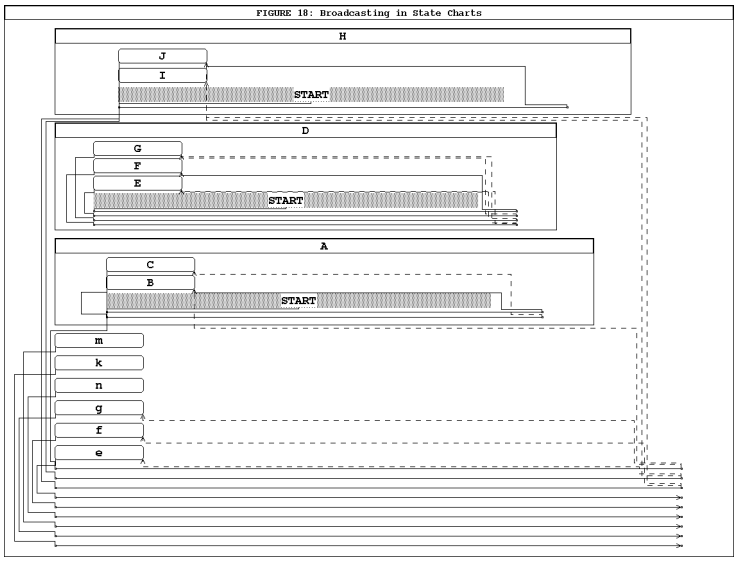


Figure 6-8: The prototype's GELO presentation of the system specified in Figure 6-7.

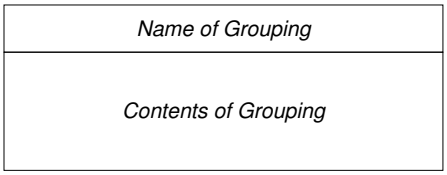


Figure 6-9: Depiction of an SPG grouping in the GELO presentation.

single step in SPG execution, and the third command causes the SPG to run until a stop node is executed.

During execution of an SPG, the GELO PM dynamically highlights nodes that are eligible for execution, nodes that are in the process of being executed, and edges along which tokens are flowing. It is thus useful not only for viewing the static structure of an SPG, but also for observing its behavior as it executes.

Development of the GELO PM yielded evidence that an SPG served well for the faithful representation of the executable semantics of a software system, that an SPG could be used as the basis for views that were less expressive than the SPG was, and that it could support interaction with users. Nonetheless, a GELO view of an SPG is hardly a realistic test of the ability of an SPG to function as the basis for interesting views. I therefore developed a second PM, this one for displaying an SPG as a statechart. My goal was to be able to automatically generate a picture of the SPG derived from Figure 6-7 in such a way that it resembled that figure fairly closely.

The statechart PM I developed supports a subset of statechart semantics. It supports the identification of SPG objects that "look like" statechart states and events, and it also generates the proper labeling of state transitions. Figure 6-10 shows how the statechart PM displays the SPG derived from the statechart of Figure 6-7. Needless to say, this machine-generated depiction does not look as nice as the manually-drawn statechart from which the underlying SPG was derived, but the important elements of the statechart are

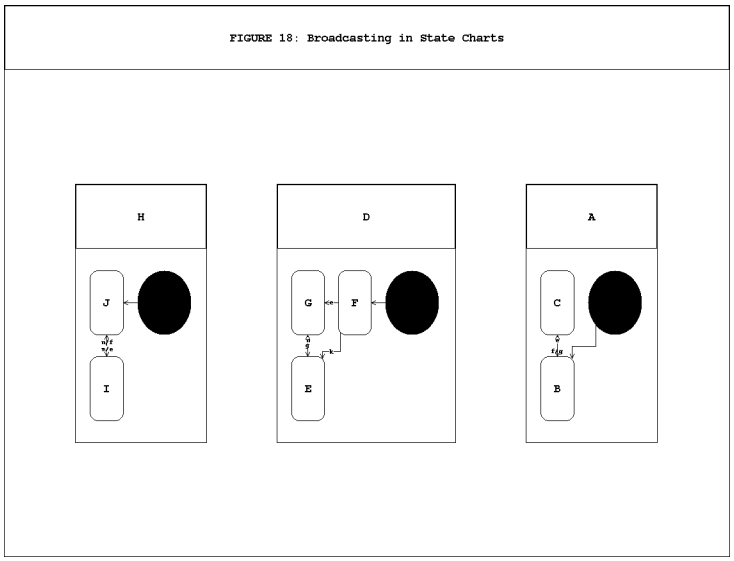


Figure 6-10: The prototype's statechart presentation of the system specified in Figure 6-7.

nonetheless clearly present. There are three top-level states (H, D, and A), and these states each contain either two or three substates.[8] Furthermore, these substates are related through event-triggered transitions, and some of these transitions (e.g., those between states I and J) themselves generate events when they (the transitions) are triggered. The primary shortcoming of this view is that it fails to indicate that each of the top-level states is orthogonal to one another. However, this would not be difficult to implement.

The statechart PM also offers all the interactive capabilities of the GELO PM, i.e., the ability to click on an object to see its annotations and to make it the current object, the ability to highlight the current object's sources and sinks, the ability to place tokens on the current object's input ports, and the ability to interactively control SPG execution by resetting the SPG, executing a single step, or running the SPG to completion.

Despite their similar interactive capabilities, the GELO PM and the statechart PM maintain separate data structures. For example, each has its own current object, and acting on the current object in one PM (e.g., to highlight its sources) has no effect on the current object in the other PM. At the same time, of course, the SPG is shared between the two PMs, so when the SPGM executes the SPG, both PMs "see" the same execution events. Similarly, when one PM requests that the SPGM execute a single step, both PMs are informed of the effects of that limited execution.

Although the GELO PM and the statechart PM are separate entities, they do contain overlapping functionality, and this common functionality is implemented by code that is shared by the two PMs. My prototype is implemented using an object-oriented programming language (C++), so each PM is modeled as a class, and the common functionality shared by the PMs is encapsulated in a base class from which both PM classes inherit. This design is shown in Figure 6-11. A particularly nice feature of this design is that it becomes quite easy to fulfill the requirement of Section 6.2 that each PM offer a prescribed

[8] The large black ovals in the figure are not states, but are instead the result of GELO's rather feeble attempt to draw the small black circles that lead to default start states in statecharts.

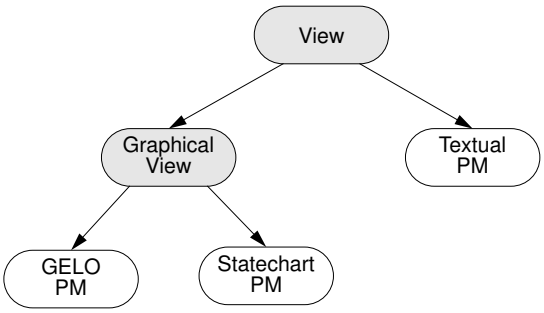


Figure 6-11: Views and presentation managers implemented in the prototype environment. Views are shaded and correspond to abstract classes in the implemenation. Presentation managers are unshaded and correspond to concrete classes in the implementation. Arrows run from superclasses to subclasses.

set of functions to be called by the SPGM. The required functions are declared virtual in the base class View, where they are defined to do nothing, i.e., to return immediately. In my prototype, all PM classes must be derived from View, and they thus need only redefine the behavior of those functions that interest them. Functions they fail to redefine exhibit the default no-op behavior.

The most interesting aspect of Figure 6-11 is that the classes in the hierarchy above the PM classes are abstract. That is, they may declare data and/or functions that are inherited by subclasses, but they themselves may not be instantiated [23]. The functionality present in the Graphical View class, for example, cannot stand on its own. It must be coupled with the additional functionality provided by either the GELO PM class or the statechart PM class before it can be used.

The difference between abstract and concrete classes in the implementation hierarchy is strikingly similar to the difference between views and presentations in my architecture for an SPG-centered MVDE. This suggests that there may be a strong relationship between views of software systems and the partial specifications of conceptual abstractions that are encapsulated in abstract classes in systems for object-oriented software development.

Figure 6-11 includes a third PM that I implemented for my prototype, one that provides only a textual presentation of an SPG. This textual presentation provides no information on the structural aspects of an SPG, but is instead devoted to the generation of trace-like information on the execution of an SPG. The presentation offers no interactive capabilities, but it does demonstrate that views providing purely behavioral information about a software system may be built atop SPGs. An example of the output from this PM when executing the SPG for the system specified by Figure 6-7 is shown in Figure 6-12.

During the course of my work on implementing the graphical PMs, it became clear that the explicit support in SPGs for view-specific annotations was of considerable utility. Both of the graphical PMs employ graph-traversal algorithms that require that they visit each node in the SPG exactly once, but because an SPG is an arbitrary directed graph, each PM must have some way of determining whether a given node has already been visited during the current traversal. It would certainly be possible for each PM to maintain a local data structure identifying the nodes it had visited, but such a data structure would require that each PM come up with a way to uniquely identify each node in the SPG; it would also

```
Moving a token on arc B-->C.
Node C has just become eligible for execution...
Starting evaluation of Event e ...
Ending evaluation of Event e.
Starting evaluation of C ...
Ending evaluation of C.
Starting evaluation of E ...
Ending evaluation of E.
Starting evaluation of G ...
Ending evaluation of G.
Starting evaluation of I ...
Ending evaluation of I.
```

Figure 6-12: A portion of the prototype's textual presentation of the execution of the system specified in Figure 6-7.

require that each PM find a way to map back and forth between its local data structure and the SPG. The availability of view-specific annotations affords a simpler solution: each PM puts a view-specific mark on each node that it has visited. When walking the graph, there is no need for a PM to map between the SPG and a local data structure, because all the information required by the traversal is located within the SPG. In essence, the ability to attach view-specific annotations to SPG components provides views with a flexible, private, and unlimited data store for each component in the graph.

6.7.1 Ancillary Research on Object-Oriented Programming

The prototype is implemented in some 9000 lines of C++, and an unanticipated by-product of my implementation work was a number of insights into object-oriented programming in general and into the specific issues surrounding C++ software development in particular.

It became apparent early on that programming tools designed for languages like C were unsatisfactory when applied to C++, so Steven P. Reiss and I collaborated on enhancements to FIELD to better support the specialized needs of C++ programmers [166]. The most notable of these enhancements was the class browser cbrowse, an entirely new FIELD tool. cbrowse provides an extensive array of information on the components of C++ software systems in both graphical and textual formats, including class names, members of classes, attributes of class members (e.g., whether they are static, their protection level, whether a function is nonvirtual, virtual, or pure virtual, etc.), and inheritance and friend relationships. Figure 6-13 shows a subset of the classes in my prototype SPG implementation as displayed by cbrowse. Arrows lead from base classes to derived classes, and virtual inheritance links are denoted with a crossbar in front of the arrowhead. Abstract classes are shown with a grey background, concrete classes with a white background.

Further analysis of the difficulties encountered when building object-oriented software with tools for conventional languages led to the realization that object-oriented programming languages, unlike procedural languages, are poorly suited for lexically based tools like text editors and grep-style pattern-matching programs [129]. This insight spurred collaborative work with Moises Lejter and Steven P. Reiss that resulted in enhancements

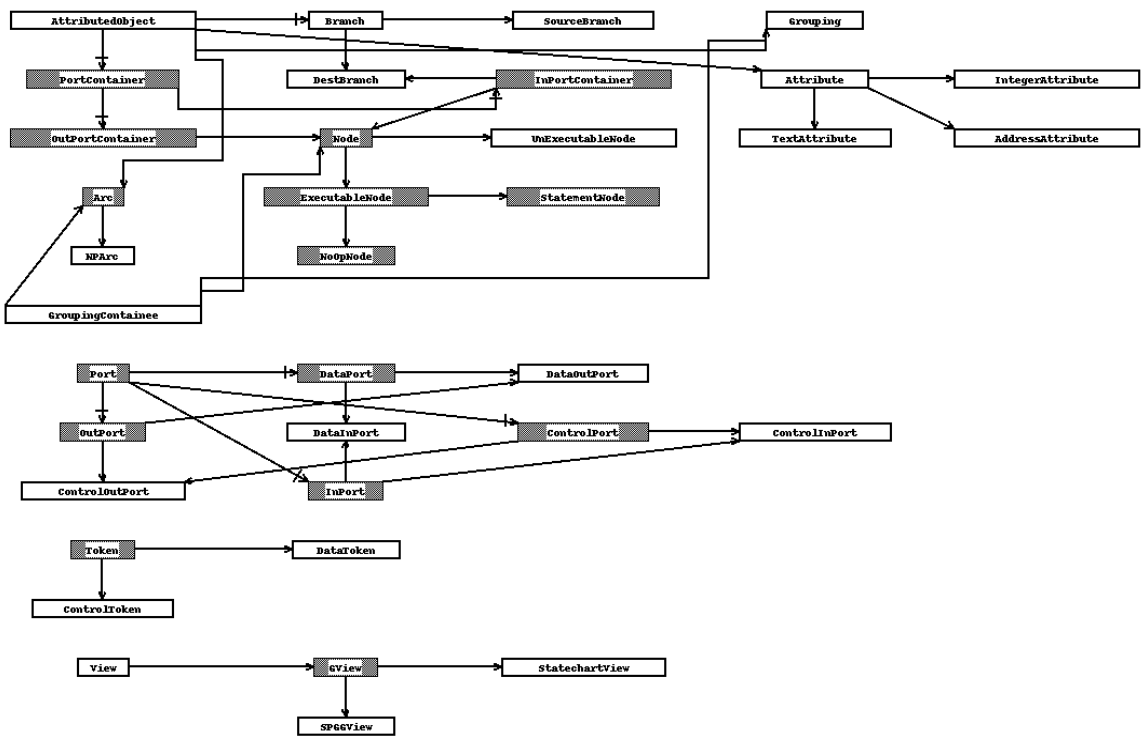


Figure 6-13: A subset of the classes in the prototype as displayed by cbrowse.

to EMACS [192] and FIELD that yielded an experimental version of EMACS that offers programmers the ability to invoke semantics-based editing commands on C++ programs [117, 118].

As a result of my implementation work on SPGs, I became interested in the problem of avoiding common but unintuitive errors in C++ programs. Moises Lejter and I identified a number of such errors, and we described how a lint-like program could automatically detect them [133]. This approach seemed promising, but it quickly became clear that the same constructs that lead to errors in most C++ programs can be legitimately employed in some restricted application domains. I therefore turned my attention to the identification of a large number of likely error conditions in C++ programs and to a characterization of the circumstances under which each was and was not an error. I eventually published the results of this investigation in book form [131].

I also began a joint effort with Carolyn K. Duby and Steven P. Reiss on the design and implementation of a metalanguage for C++ programs that allowed programmers to specify a wide range of application-specific constraints such that violations of the constraints would be automatically detected [54, 132]. The resulting metalanguage, CCEL (the C++ Constraint Expression Language), has since grown into an independent and ongoing research project in its own right.

6.8 Summary

My design for an SPG-based MVDE is to encapsulate the SPG as an abstract data type, with the SPGM acting as the interface to the type. Modifications to the SPG are initiated by PMs, which are themselves encapsulated and which translate user actions on their presentations into SPGM operations. The SPGM then notifies PMs of changes to the underlying SPG, and the PMs update their presentations accordingly.

As encapsulated entities, both Presentation Managers and the SPG Manager have well-defined interfaces through which all interaction takes place. The SPGM interface includes routines that allow PMs to perform arbitrary traversals of the SPG; to add, modify, and remove components of the SPG; and to initiate, terminate, and single-step through execution of the SPG. The SPGM requires that each PM interface allows the SPGM to notify the PM when certain semantically meaningful actions take place within the SPG. Such actions include the addition, modification, and/or removal of components to/from the SPG, as well as information on the progress of execution within the SPG.

When performing presentation updates, PMs may adopt either batch or incremental strategies, or they may employ a combination of the two. Some SPG updates are more difficult to display than others, and PMs must be prepared to cope with semantics mismatches between the language they present and the features supported by SPGs. One way to handle such mismatches to to employ black boxes in the presentation.

The detection of programming errors is a task that must be shared by the SPGM and PMs, with PMs performing view-specific error detection at the time they create and/or modify a part of the software system under development. The SPGM is responsible for view-invariant error detection, which is primarily concerned with ensuring the topological and execution-time integrity of the SPG.

The sample views in Table 6-1 suggest the breadth of views that SPGs can support. They indicate that views providing information on a system's static execution flow, its system structure, and/or its dynamic execution can all be implemented in terms of SPGs, and they show that an SPG-based MVDE can offer a broad range of views of software systems, including some that are only minimally related to the physical "source code" of the system.

As part of my research for this dissertation, I developed a preliminary prototype implementation of an SPG-based environment, including the creation of three PMs, each of which offer dynamic visualization of program execution. The prototype offered valuable experience with the ideas behind SPGs, and it also proved to be a effective research vehicle for analyzing the software development needs of C++ programmers.

Chapter 7

A Preliminary Assessment

In the chapters prior to this one, I have motivated my research on multiple-view software development, justified my decision to focus on a canonical representation of software systems as the mechanism for achieving environment integration, described the shortcomings of existing canonical representations, defined SPGs, described how they can be used to solve important representational problems, and explained how they fit into an architecture for an MVDE. In this chapter, I provide a preliminary assessment of the strengths and weaknesses of SPGs in terms of the two primary constraints on a canonical representation that I introduced in Chapter 3: expressiveness and generalized invertibility.

7.1 A Paper Experiment

The basis for my assessment is a paper experiment in which I developed detailed mappings between SPGs and three views. Two of these views, deterministic FSAs and a subset of Pascal, are static read-write views; they thus comprise two mappings each (from the view to an SPG and from an SPG to the view). The third view, a dynamic runtime stack (see Section 6.6.4), is inherently read-only. It therefore consists of only a single mapping (from an SPG to the view). The experiment therefore involved the development of five algorithms:

- Translation of an FSA into an SPG.
- Translation of an SPG into an FSA.
- Translation of a Pascal program into an SPG.
- Translation of an SPG into a Pascal program.
- Translation of an executing SPG into a runtime stack.

Figure 7-1 depicts the relationships among an SPG, these views, and the mappings between them.

An additional component of the experiment was to employ a multiparadigm development methodology in the creation of a new program. This new program was required to be first partially developed using one view, then to have its behavior altered through a second view. The function of the program I wrote was to recognize unsigned real numbers in Pascal. Such

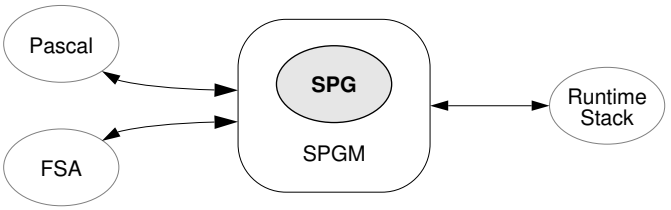


Figure 7-1: The relationships among an SPG, the views, and the view-SPG and SPG-view mappings involved in the experiment.

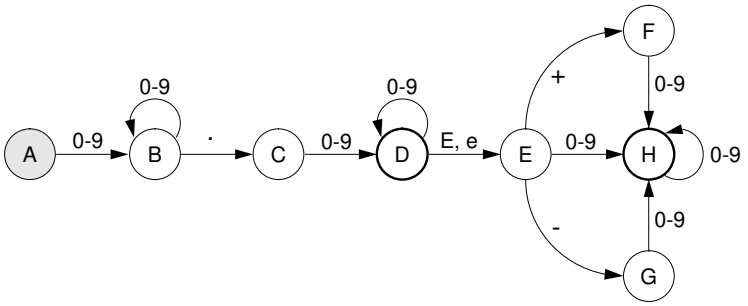


Figure 7-2: An FSA for recognizing unsigned real numbers. The shaded state is the initial state, and states with heavy outlines are accepting states.

numbers consist of a mandatory real part followed by an optional exponent part. The input to the program is read from the standard input.

For the purpose of the experiment, I made the assumption that the characters period (“.”), E (“E” or “e”), plus sign (“+”), and minus sign (“-”) are integers, because SPGs currently allow only integers as data. This is purely an assumption of convenience, because each of these characters could be encoded as a unique integral value. (In a language like C, in fact, there is no need to make this kind of assumption, because it is effectively part of the language definition.)

The experiment began with my writing FSA and Pascal programs to recognize unsigned real numbers. These programs are shown in Figures 7-2 and 7-3, respectively. The structure of the FSA program is unsurprising, but that of the Pascal program may not be. Despite decades of experience with the creation of procedural programs for implementing FSAs, the question of how to properly code an FSA can still generate controversy [101, 119]. Compiler texts tend to encourage and compiler-building tools like lex invariably employ a table-driven approach to the problem, but for the purposes of this experiment, I used a naive translation of the topology of the FSA into Pascal control structures.

Each of the sections that follows corresponds to one of the mappings I developed for this experiment. I begin each section with a general description of my algorithmic approach to the mapping being discussed; readers interested in the details of any particular mapping are referred to Appendix B, where I provide complete pseudocode programs for all the mappings. Next, except for the runtime stack view, I show the results of applying the mapping to the FSA program, the Pascal program, or SPGs derived from those programs. Finally, I discuss the insights I gained during the development of the mappings, and I examine the implications of these insights on the overall suitability of SPGs as a basis for

```
program fsa(input, output);

  var c: char;

  procedure error;
  begin
    writeln('Error!');
  end;


  begin
  read(c);
  if (c < '0') or (c > '9') then error;

  repeat
    read(c);
  until (c < '0') or (c > '9');

  if (c <> '.') then error;

  read(c);
  if (c < '0') or (c > '9') then error;

  repeat
    read(c);
  until (c < '0') or (c > '9');

  if (c = 'e') or  (c = 'E') then
    begin
    read(c);
    case c of
      '+':        read(c);
      '-':        read(c);
      otherwise   ;
    end;

    if (c < '0') or (c > '9') then error;

    repeat
      read(c);
    until (c < '0') or (c > '9');
    end;

  writeln('Okay.');
  end.
```

Figure 7-3: A Pascal program to parse unsigned real numbers.

multiple-view software development.

For the purposes of this experiment, I confined myself to batch-mode translation algorithms, rather than incremental versions (see Section 6.3). I also limited my efforts to the development of "minimal" algorithms, i.e., mappings that perform well on graph topologies that are expected to be common, but that do not necessarily produce useful results on all possible graph topologies. As such, my work in this chapter may be characterized as that of a developer who is interested in putting together a set of "quick and dirty" mappings between SPGs and some views of interest. Though not definitive, this kind of experiment is far from trivial, and it is certainly demanding enough that it provides a valid basis for a preliminary assessment of the utility of SPGs as a basis for translating between views of software systems.

The mappings that follow are into and out of *views*, not presentations. Because views correspond to abstract syntax and presentations correspond to concrete syntax (see Section 1.1), each mapping assumes that its input has been predetermined to be syntactically and semantically valid. As a result, the FSA-to-SPG translator doesn't check for an invalid FSA, and the Pascal-to-SPG translator doesn't check for an invalid Pascal program.

7.2 Translating an FSA into an SPG

The Input

An FSA is a directed graph consisting of one or more connected components. Each connected component may be named, may have at most one initial state, and may have zero or more final states. Each state in the FSA has an associated action routine that is invoked automatically when the state is entered. The action routine is specified by name. If no action routine is specified, an FSA-specific default routine is invoked. If no default routine has been specified for an FSA, a no-op routine is called.

Input values controlling state transitions may be specified in the form "3" or "3, 8, 10" or "3-5, 1, 22-25". That is, each transition is controlled by a comma-separated list of single values or value ranges. The input sets enabling distinct arcs leaving a state must be disjoint, because each FSA must be deterministic.

When an input character is encountered that does not match any of the current state's outgoing arcs, an error routine is invoked. The name of this routine may be specified. If there is no such specification, a default routine that issues an error message and terminates program execution is called.

The Approach

When mapping from an FSA to an SPG, I consider each connected component of the FSA separately, and I map each component into a distinct Subprogram grouping. If a connected component contains a start state, I insert into the SPG a Call node that invokes the subprogram corresponding to the FSA component, and I make this Call node a start node.

Each FSA state maps into an SPG Call node; the routine invoked corresponds to the action routine for the state. If no routine is specified, a call is made to the routine specified

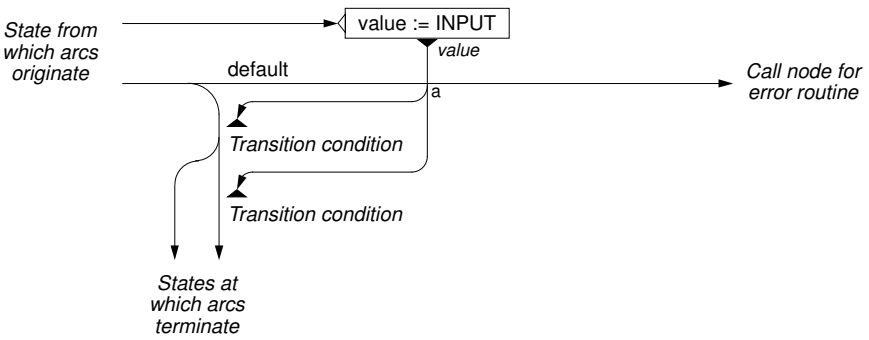


Figure 7-4: The SPG subgraph into which an FSA arc is translated.

as the default action routine. If there is no such routine in the FSA, a new no-op routine is generated, and that routine is designated the default action routine.

The name of each state could easily be stored in the generated SPG as the value of a Name annotation on an unexecutable grouping that contains the Call node corresponding to the state, but in the interest of adhering to the philosophy of minimal mappings, I chose not to generate such a grouping. As a result, the name of a state is lost during the translation into an SPG.

Each arc leaving an FSA state maps into an SPG subgraph consisting of several components. One component is a Multiassignment node that reads an integer from standard input and passes it along to the branch conditions of an edge that leads from the state to other states. A second component is this edge, which, in addition to containing branches leading to the endpoint of each arc, also contains a destination branch leading to a Call node for an error routine. This latter destination branch has the condition default and is traversed whenever invalid input is encountered. A third component is an edge leading from the Call node representing the FSA state to the Multiassignment node representing the reading of an integer from the standard input that takes place at that state. Figure 7-4 shows the relationship of these components.

If an arc both originates and terminates at the same state, a Loop grouping is added to the SPG containing both the Call node corresponding to the state and the edge components corresponding to the arc. The destination branch leading back to the Call node comprising the state is given an annotation identifying it as the Loop Return part of the loop.

Complete pseudocode for the mapping from an FSA to an SPG can be found in Section B.1. Application of this mapping to the FSA of Figure 7-2 yields an SPG containing the Subprogram grouping shown in Figure 7-5. The complete SPG also contains Subprogram groupings for the default action and error routines, as well as a Call node to invoke the subprogram shown in Figure 7-5, but these have been omitted from the figure due to space limitations.

Observations

The most difficult part of the development of this mapping had nothing to do with achieving the correct executable semantics for an FSA. Creating an SPG with the correct executable semantics is trivial, i.e., there is no expressiveness problem. What is distinctly *un*trivial is trying to figure out in more general terms what an FSA really *means*. Computationally,

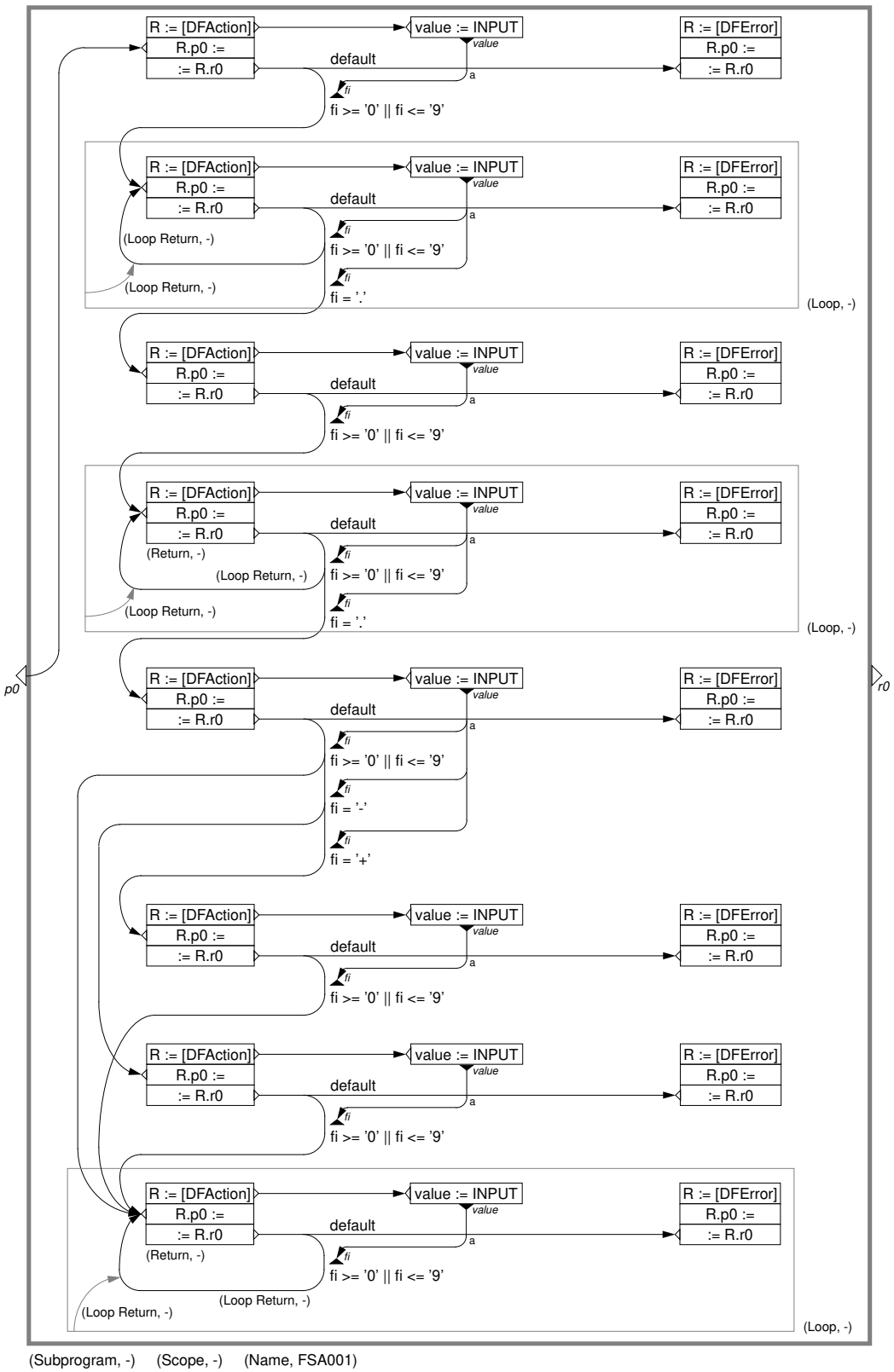


Figure 7-5: The primary Subprogram grouping in the SPG resulting from the FSA of Figure 7-2. Some routine and port names have been abbreviated to save space.

what does a state represent? What does a transition represent? These are the truly troublesome issues that confront the designer and implementer of a mapping from a high-level abstract view like an FSA into the procedural, computational domain of an SPG.

Because an FSA is so simple, the resulting SPG can be severely underconstrained. Under such conditions, it is not at all clear what the appropriate philosophy is regarding the creation of SPG components. For example, the FSA-to-SPG translator may generate two routines, one as a default action routine, one as the routine to invoke when an input error is detected. Neither of these routines requires any local variables, so should the FSA declare the resulting Subprogram groupings to be scopes? If not, many views will have difficulty in making sense of the routines, because it is common for routines to also be scopes. Of course, adding gratuitous scopes is likely to be painful to the developers of views that don't support subprograms as scopes. In the case of this particular issue, such pain is unlikely — most every language allows for the declaration of local variables — but the general question persists: should views create an SPG that corresponds exactly to what they need, or should they try to do a more or less "complete" job that reflects more common programming conventions? The idea of "complete," of course, is view-dependent, and that is precisely the problem.

In a similar vein, note that a connected component of an FSA may be anonymous. When a Subprogram grouping is generated for a component, then, what should be done about a Name annotation? It could be omitted, of course, but that is likely to cause problems for views that have no notion of anonymous routines. The only solution to the problem would be for all views to contain code for assigning names to anonymous subprograms, but such code duplication was one of the motivating factors for the development of a canonical representation in the first place! A better solution is for the FSA view to generate a name for an anonymous component (or to query the programmer for a name), but this cannot be enforced by the SPGM, it can only be followed as a convention. Unfortunately, views that allow anonymous routines (e.g., lambda functions) are likely to find the convention unintuitive and unnatural.

7.3 Translating Pascal into an SPG

The Input

The mapping I developed accepts as input a subset of Pascal. The subset includes the following features:

- Variable declarations (for scalar integers only).
- Procedures and procedure calls.
- Parameters: formal and actual (scalar integers only); pass-by-value (i.e., non-var) only.
- Assignment statements.
- Compound statements (i.e., begin-end blocks).
- Conditional statements (i.e., if-then-else statements).
- case statements, including the common otherwise extension.

- Looping constructs: repeat-until statements and for loops.
- Empty statements.
- I/O: read statements (excluding a file specification) and write and writeln statements (including literal strings, but excluding file specifications).
- The conventional relational, arithmetic, and logical operators.

It excludes these features:

- Program parameters (i.e., file specifications for input and output).
- Functions and function calls; var-parameters.
- Forward declarations.
- Nested procedures.
- Type declarations (including arrays, enums, records, sets, files, and pointers).
- Declarations of constants and labels.
- goto, while, with, and readln statements.
- The operators in, div, and mod.
- Comments.
- Embedded quotes in literal strings.
- The following predefined functions and procedures: abs, cos, trunc, sqr, arctan, round, sqrt, ln, sin, exp, ord, chr, succ, pred, odd, eoln, eof, rewrite, reset, put, get, page, new, dispose, pack, and unpack.

The features that are excluded are essentially similar to other features I have included (e.g. functions, while statements), pertain to data modeling (e.g., type declarations, with statements), or simply make the example here more tractable (e.g, nested procedures, comments).

The Approach

Translation of a Pascal program into an SPG is straightforward and can be accomplished in a single pass over the program. The program itself becomes a Scope grouping (the global scope), the variable declarations at global scope become Declaration nodes within this grouping, and procedures become Subprogram groupings within this outermost grouping. Within either a procedure or the main block, statements are translated in the order in which they occur in the manner suggested by the pseudocode routine of Figure 7-6.

Complete pseudocode for the mapping from a Pascal program to an SPG can be found in Section B.2. Application of this mapping to the program of Figure 7-3 yields the SPG shown in Figure 7-7.

```
procedure translateStatement(Statement S, Grouping CG, DestinationBranch CDB)
    case (type of S) of:
      compound-statement:
        Create a new unexecutable grouping UG in CG;   // for the block
        translateBlock(UG, CDB);
      conditional-statement:    translateConditional(CG, CDB);
      case-statement:           translateCase(CG, CDB);
      repeat-until-statement:   translateRepeat(CG, CDB);
      for-statement:            translateFor(CG, CDB);
      procedure-call-statement: translateCall(CG, CDB);
      assignment-statement:     translateAssignment(CG, CDB);
      read-statement:           translateRead(CG, CDB);
      write-statement:          translateWrite(CG, CDB, false);
      writeln-statement:        translateWrite(CG, CDB, true);
      empty-statement:          translateEmpty(CG, CDB);
    end;
```

Figure 7-6: Central dispatch routine in the Pascal-to-SPG translator of Section B.2.

Observations

The fact that Pascal requires that variables and procedures be declared before use simplifies the process of generating an SPG, because a simple linear traversal through the Pascal source is sufficient to generate the SPG without having to worry about references to variables and/or routines that have not yet been declared.[1] In addition, translation of Pascal assignments, calls, and output statements is facilitated by the fact that the right-hand side of each assignment in a Multiassignment node is itself an expression. As a result, there is no need to encode in an SPG subgraph the evaluation of the expressions making up the actual arguments to a procedure call. The Pascal-to-SPG translation would not be quite so straightforward if the Pascal subset I accepted included function calls, but under no conditions would a view-to-SPG translator have to deal with the tedium of constructing expression trees for simple expressions. The expressive power of SPGs is more than sufficient to accommodate the semantics of Pascal constructs, so there is no difficulty in accurately representing the executable behavior of a Pascal program. As a result of all these factors, development of a Pascal-to-SPG translator is not a particularly demanding task.

At the same time, it is not without its unpleasant aspects. Generating the portion of the SPG that converts data tokens arriving on InOut ports into initialization values of local parameters (see Section 4.3.3) is awkward. However, it is not clear how the definition of an SPG could be modified to simplify this task, because the initialization subgraph is not needed for dataflow languages — the InOut port itself suffices. One might reasonably expect common operations (such as parameter initialization) to be facilitated through a support library of useful SPG-construction routines (see also Section 3.5.4). In the presence of such library routines, one could reasonably expect the effort needed to develop view-to-SPG

[1] This is not the case with an FSA, where, for example, a state may have as its action routine an FSA that has not yet been translated into an SPG. (For details, see Section B.1.) Such complications are far from insuperable, but they do complicate the process of writing view-to-SPG mappings.

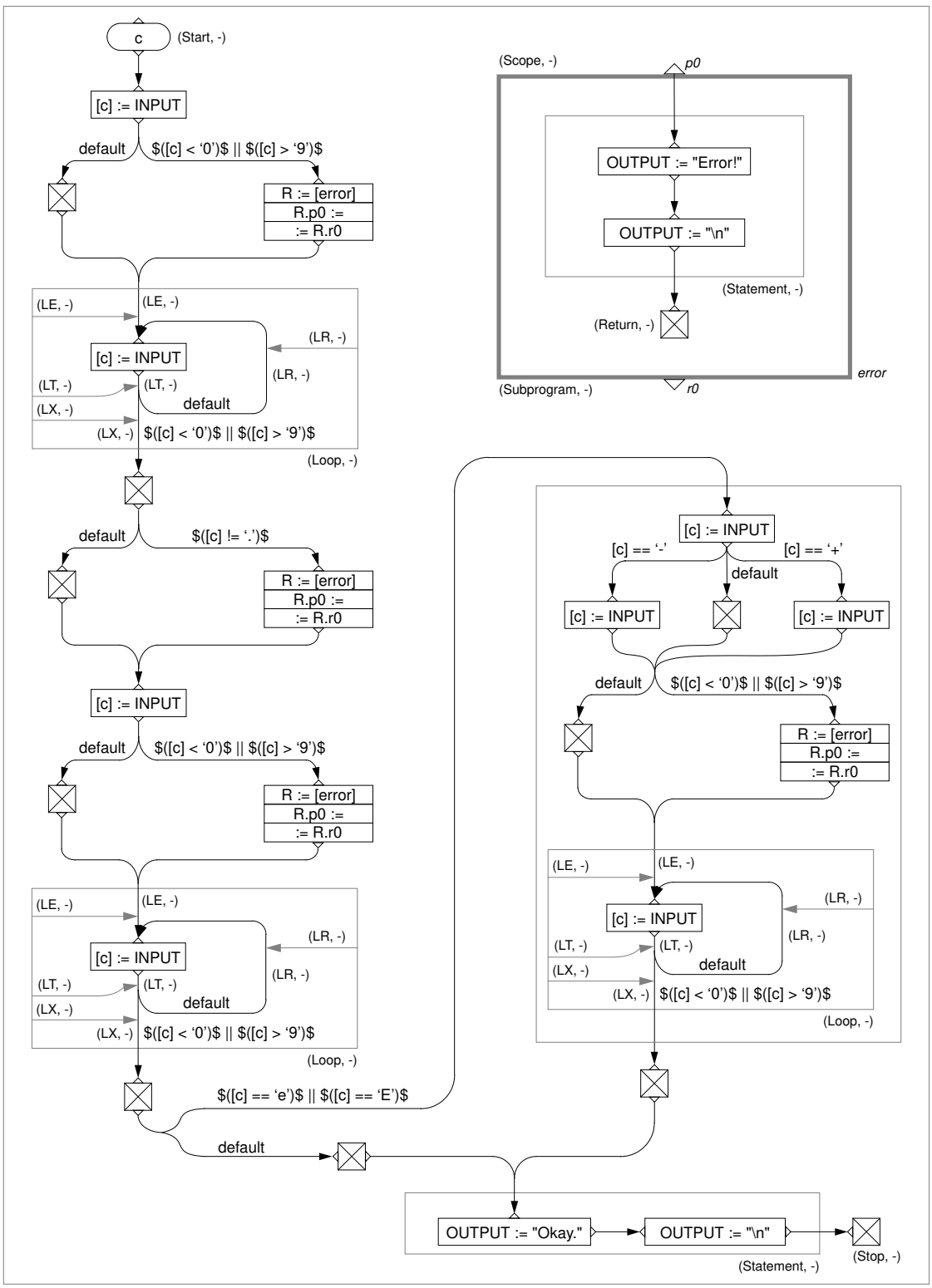


Figure 7-7: The SPG resulting from the Pascal program of Figure 7-3. Names of loop annotations are abbreviated as follows: LE = Loop Entrance, LR = Loop Return, LX = Loop Exit, LT = Loop Test.

mappings for views employing common semantic concepts to be considerably reduced.

Unsurprisingly, generating an SPG for Pascal is in many ways similar to code generation for a Pascal compiler, but there is less flexibility. During generation of object code, any set of instructions that yields the correct execution-time behavior is acceptable. If it is convenient for the code generator to introduce behaviorally neutral code-generation artifacts into the emitted code, there is no reason not to do it. For example, it might be convenient to have exactly one exit location in the code generated for each procedure, so a compiler might have all logical exits jump to the single physical exit in the generated code. The jumps would not be present in the original source code, however — they would merely be artifacts of the code generation process. When translating into an SPG, such artifacts must be avoided, because an SPG must accurately represent "what is going on," and artifacts obscure what is going on. Generating an SPG, then, is in some ways a more tightly constrained task that is code generation for a compiler. Of course, code generation strategies in a compiler are strongly influenced by considerations of runtime performance, and these considerations are absent for view-to-SPG translators. The bottom line is that code generation and SPG generation, though sharing some common features, are in other ways fundamentally dissimilar.

7.4 Translating an SPG into an FSA

The Input

In my SPG-to-FSA translator, I consider a software system represented by an SPG to be made up of a number of separate FSAs, one for each of the following SPG components:

- A Subprogram grouping.
- An STask grouping.
- A Rendezvous grouping.
- The SPG subgraph consisting of the components not contained in any of the groupings above, i.e., what would be called the "main block" of a Pascal program.

As a result, the algorithm I developed takes as input *one* of the components above, and an entire SPG will typically yield multiple FSAs.

The Approach

The translation proceeds in three stages. First, it identifies the SPG subgraphs that "look like" FSA arcs. Second, it identifies the SPG subgraphs that "look like" states, including start states and accepting states. Finally, it identifies the action routine associated with each FSA state. The algorithm does not attempt to identify a name for the states of the FSA, but it would not be difficult to augment the translator to look for groupings of the kind I discussed in Section 7.2.

My algorithm for arc identification is based on looking for locations in the SPG where unprocessed input data is used to control token flow along the branches of an NP_1 or a Select edge. There are two general topological forms to look for in the SPG, one based on data flow, one based on control flow. Both forms consist of a Multiassignment node that

reads from INPUT and uses the result of the read to control token flow on a nearby edge. In the data flow form, the datum read from INPUT is placed on an NP_a data edge such that each of the destination branches of that edge terminates at a branch condition of the nearby edge. In the control flow form, the datum is stored in a variable that is then used in branch conditions of one or more nearby edges. Readers interested in a more precise description of the recognized topologies are referred to Section B.3.

When unexpected input is encountered in an FSA, the machine follows an implicit transition into an implicit and inescapable error state, but in real programs, such transitions and states are explicit. As a result, an SPG-to-FSA translator is faced with the task of distinguishing between input-dependent actions that correspond to FSA arcs and similar-looking input-dependent actions that correspond to implicit transitions to an FSA error state. My translator employs the heuristic that edge branches followed when input data falls in a *constrained range* lead to valid (and thus explicitly represented) FSA states, while branches followed when input data falls in an *unconstrained range* (i.e., a conceptually infinite set of values) lead to invalid (and thus implicit) FSA states. For example, given an SPG representation for this fragment of Pascal,

```
read(i);                    { assume i is declared to be an integer }
if 0 <= i and i <= 9 then
  ...                       { i's range is constrained to [0,9] }
else
  ...                       { i's range is unconstrained: }
                            { anything outside [0,9]      }
```

the algorithm will conclude that the then arm of the conditional corresponds to an explicit transition to a valid FSA state and that the else arm corresponds to an implicit transition to the error state.

Finding the SPG subgraph corresponding to an FSA arc is complicated somewhat by the fact that, in the control-flow form, a single FSA arc may be represented by multiple edge conditions. The algorithm therefore recognizes that nested conditionals can actually correspond to a single conceptual FSA arc. For example:

```
read(i);
if 0 <= i
  begin           { i's range is unconstrained:  anything non-negative }
  if i <= 9 then
    ...           { i's range is now constrained to [0,9] }
  end
else
    ...           { i's range is unconstrained: anything negative }
```

I adopt the same arc-labeling convention for my SPG-to-FSA translator as I did for my FSA-to-SPG translator, namely, that arc labels must consist of a comma-separated list of single values or value ranges. These ranges must be statically evaluable, so, for example, the range "x-y" (where x and y are variables) is not allowed.

Identification of FSA states proceeds by starting at arc endpoints (i.e., at the endpoints of SPG subgraphs corresponding to FSA arcs) and then searching both forward and backward along executable edges (but not along any SPG subgraphs that correspond to FSA arcs) until the largest possible subgraph has been identified. If, during this search, subgraphs for two distinct states "bump into" one another, the states are merged, if possible. An SPG subgraph G may correspond to an FSA arc that both originates and terminates

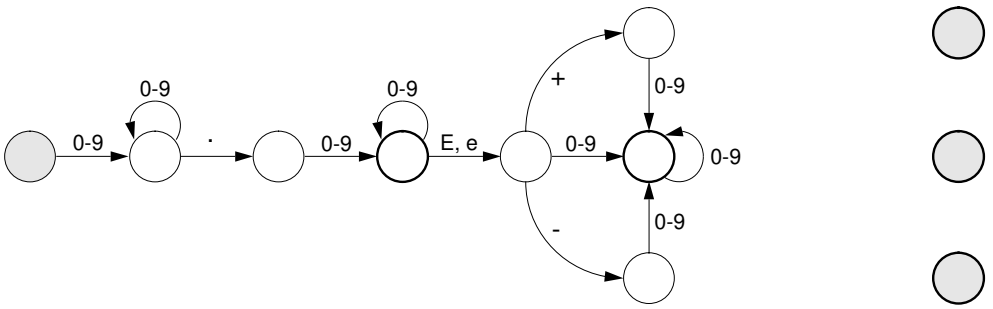


Figure 7-8: The FSA resulting from the FSA-derived SPG of Figure 7-5.

at the same FSA state S only if both G and the SPG subgraph corresponding to S are contained inside the same Loop grouping. This restriction prevents an entire SPG from being collapsed into a single FSA state with all arcs leading both from and to that state.

If the SPG subgraph being viewed as an FSA contains exactly one start node N, and if N is part of a subgraph corresponding to an FSA state S, S is a start state. If the subgraph being viewed as an FSA is a Subprogram grouping, it cannot contain any start nodes, so a different algorithm is employed. For a Subprogram grouping, if there exists a node that must be the first node executed whenever the subprogram is invoked, the FSA state that contains that node (if there is one) is a start state. If the SPG subgraph corresponding to an FSA state contains a Return node and/or a Stop node, it is a final state.

The action routine for each state is determined by inspecting the first node encountered along any arc when entering that state. If all such nodes are Call nodes that invoke the same routine, that routine is the action routine for the state. Otherwise the action routine is undefined.

Complete pseudocode for the mapping from an SPG to an FSA can be found in Section B.3. Application of this mapping to the SPG of Figure 7-5 yields the automata shown in Figure 7-8.[2] It is clear from the figure that the set of FSAs generated from the SPG derived from the original FSA contains one FSA that is identical to the original. The additional single-state FSAs correspond to the Subprogram grouping for the default action routine, to the Subprogram grouping for the error routine, and to the Call node that invokes the subprogram corresponding to the original FSA. Judicious use of view-specific annotations during SPG generation would make it easy to suppress these superfluous FSAs during translation back to an FSA view, thus yielding a single FSA that is identical to the original.

Applying the SPG-to-FSA translator to the Pascal-derived SPG of Figure 7-7 yields the automata shown in Figure 7-9. Despite the fact that the SPG from which it was generated is not at all similar to the one giving rise to the FSAs in Figure 7-8, the results are nearly identical, the only substantive difference begin the lack of final states in the primary FSA of Figure 7-9. This omission can be traced to the facts that, first, each Pascal procedure must have exactly one exit, and second, there is no construct in Pascal that causes program execution to terminate immediately. As a result, the heuristic employed by my SPG-to-FSA

[2]When examining Figures 7-8 and 7-9, bear in mind that all *presentation*-specific aspects of the FSA (e.g., layout of states and arcs) is produced not by the translation algorithm (which maps an SPG into an FSA *view*, not an FSA presentation), but was manually introduced after the translation was complete.

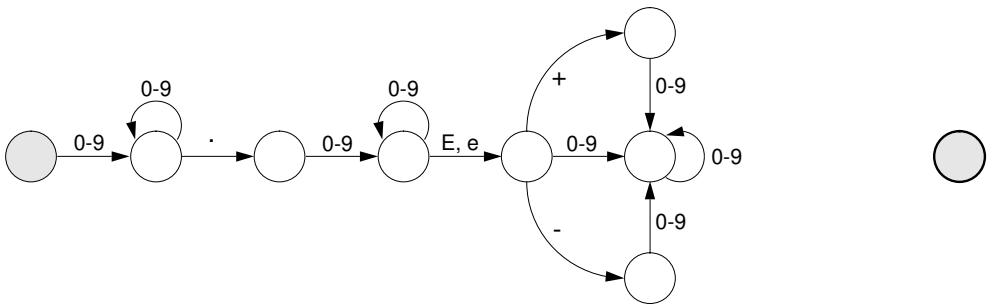


Figure 7-9: The FSA resulting from the Pascal-derived SPG of Figure 7-7.

translator fails to identify the appropriate states as final states.

Observations

The algorithm I use in my SPG-to-FSA translator assumes that the structure of the FSA can be deduced by mimicking the structure of the flow of control through a program. That is, it assumes that FSA states can be discovered through examination of a program's control structures alone, that such states are not encoded in data structures or as the value of variables, etc. I pointed out in Section 7.1 that this assumption is suspect, at least for the FSAs that are implemented for tasks such as lexical analysis. However, it may not be an unreasonable assumption for gaining insight into the structure of general programs, especially if the notion of an arc (i.e., the heuristic used to identify one) is generalized to something more than a branch based on reading from standard input. This issue — that of knowing what to look for in an SPG to recognize the existence of an FSA component — is essentially the inverse of the problem I discussed earlier: what should it mean to view a program as an FSA? Without a well-defined answer to this higher-level conceptual question, it is meaningless to attempt to evaluate how well or how poorly SPGs allow one to implement the answer.

The algorithm I use for mapping an SPG to an FSA is really almost two algorithms in one, because the translator must duplicate a fair amount of conceptual effort (though not code) to deal with SPG subgraphs that "look like" FSA arcs based on data flow and with other subgraphs that "look like" FSA arcs based on control flow. Furthermore, the checking of the interconnections between nodes, edges, and branch conditions is quite tedious. I do not believe this is reflective of a flaw in SPGs, but rather is indicative of the fact that going from a relatively low-level representation like SPGs to a relatively high-level abstraction like FSAs is in general a nontrivial task. If the patterns in the graph that correspond to what we want to see as FSA arcs are complex, it is not surprising that the code to look for the patterns is complicated, too. Nonetheless, it is clear that the translator implementation is no "simple" task, and in that sense it can be argued that SPGs have fallen short of my (perhaps overly ambitious) goal to facilitate the development of easy-to-write mappings to all kinds of views.

It is worth noting that an FSA resulting from my translator may yield misleading results because, given two states A and B with no arc between them, it need not be the case that there is no way to get from A to B or from B to A. It will only be the case that there is no way to get from one state to the other *based purely the result of a read from standard input.*

In other words, there might still be ways to achieve a state transition that do not involve reading from standard input.

It is possible that one could develop an algorithm that would show an FSA view of an SPG in such a way that if two FSA states were distinct and connected by an arc, the *only* way to get from one state to the other would be by traversing that arc, but that would be a different interpretation of what it means to view a general-purpose program as an FSA. There being no obvious answer to that question, it is not clear which conceptual SPG-to-FSA algorithm is the "right" one.

Examination of the results of this algorithm makes clear that there is not necessarily any simple relationship between an SPG component and its corresponding view component(s). For example, some SPG components have no view counterpart. In this mapping, SPG subgraphs corresponding to the implicit FSA error state may never be assigned to any FSA component, because unconstrained arcs are never used as the starting point for a search for a state. As a result, if an SPG subgraph is accessible only via unconstrained arcs, it will never be searched. On the other hand, it is possible for a single SPG source branch to be part of any number of FSA arcs, because each destination branch of an edge may yield a different arc. An SPG component may therefore correspond to zero, one, or many entities in an FSA view of that SPG.

7.5 Translating an SPG into Pascal

The Input

The translator I developed works with any valid SPG.

The Approach

My translator is not tremendously ambitious in its attempt to translate an SPG into Pascal. It is designed to work with SPGs that correspond to the model of computation embodied by Pascal, i.e., serial, deterministic, based on control flow. For the most part, when it sees something that fails to fit this model, it produces a comment in the generated Pascal to the effect that further translation of the routine (or the main block) is impossible, nd it discontinues translation of the grouping containing the offending topology. Sometimes it adopts a less Draconian strategy. For example, when it encounters STasks and nested routines, neither of which is in the subset of Pascal I support in this experiment, the translator produces comments containing the names of the entries or routines, respectively.

The result of the translator is in fact not so much a Pascal program *per se* as a program that looks very much like Pascal. There is, however, no guarantee that it is a *valid* Pascal program. For example, a legal SPG may have more than one variable or subprogram with the same name in a Scope grouping, but this is illegal in Pascal. My translator does not check for such conditions. In fact, I believe that such activity is probably best left to a separate "Pascal checking" view. Such a view would have a different purpose from the one I implemented, which has as its purpose to to produce an accurate view of the semantics of the program represented by an SPG and to present these semantics, as much as possible, in Pascal's syntax.

My general approach to the translation problem is straightforward. The subset of Pascal I considered in this experiment yields programs that begin with variable declarations, continue with procedure declarations and bodies, then conclude with the body of the main block, so my translator first translates Declaration nodes at global scope, then translates Subprogram groupings nested inside the global scope, and finally translates the SPG subgraph corresponding to the code inside the main block.

When generating executable statements, the algorithm iteratively translates an SPG node, then examines the edge[3] emanating from that node. In general, if the edge has a single destination branch, that branch is followed and the node it leads to is translated. If the edge has two destination branches, an if-then-else statement is generated, and each of the two nodes led to by the edge is translated. Finally, if the edge has more than two destination branches, the conditions on the destination branches are examined to see if they are compatible with a case statement. If so, a case statement is generated, otherwise a series of if-then-else statements is produced.

This high-level sketch of the algorithm ignores a number of complications, including the detection of loop entry and exit. The latter are structurally identical to conditionals (except for loop-specific annotations), so it is important to check for a loop exit before considering an if statement. The algorithm must also detect join-points after conditionals and loop entries, because the code downstream from a branch in the flow of control must be generated only once, and only in the right place. Details on these points and on every other aspect of the SPG-to-Pascal mapping I implemented can be found by consulting the (commented) pseudocode in Section B.4.

In order to facilitate the production of a consistent concrete syntax for a program over repeated SPG-to-Pascal translations, the translator adds a global grouping and assigns it a name if it can't find an existing global grouping that has a name. In the comments in the pseudocode for the translator (see Section B.4) I discuss the possibility of also adding to the SPG a set of unexecutable edges with a Precedes annotation so that variable declarations would always be written out in a consistent order, but the translator as implemented does not do this.

The translator writes out procedure declarations in a random order; it does not take into account the fact that Pascal requires that procedures be declared before use. Adding the necessary dependency analysis to determine an acceptable declaration order is not a lot of work (one need only examine all Call nodes in a Subprogram grouping and see which routines they call), but, in the spirit of a "quick and dirty" implementation, my algorithm fails to do it. It cuts additional corners in the following ways:

- The translation of if-then-else, case, and for statements always yields begin-end blocks, even for a single statement. It is possible to avoid this with more ambitious translation algorithms that perform lookahead to determine whether a single statement or a statement sequence is to be translated. Similarly, the algorithm may not get the semicolons in nested conditional statements exactly right, but, with a little more bookkeeping, it would not be difficult to do.

- The algorithm doesn't approximate simple non-strict procedure calls with the concrete syntax for strict calls, but this, too, would not be difficult to add to the translator. The

[3] In general, my translator insists that each node have exactly one executable output port. If a node has more than one such port. the algorithm throws up its computational hands and refuses to continue.

general approach would be to afford special treatment to edges receiving a data token carrying the value of an RIID from a Call node to Parameter nodes. The parameters passed in the Pascal translation of the procedure call would correspond to the union of the data tokens passed to the invoked routine from the Call and Parameter nodes.

- The algorithm refuses to translate portions of an SPG that read from INPUT and place the datum read directly onto a data port or onto OUTPUT. Generation of the Pascal corresponding to such constructs is not difficult to do, but it requires the introduction of a new variable (to be used as the argument to **read**) that is not currently used in the program. This would conflict with the current on-the-fly translation algorithm, because the current algorithm has already emitted Pascal for all Declaration nodes before it translates the remainder of the SPG.

- The translator ignores Statement groupings.

The above limitations tend to *restrict* the Pascal that the translator will produce. That is, if the translator encounters one of the constructs above, it may produce a sub-optimal translation or it may refuse to translate a construct altogether, but in no case will it generate Pascal that is unfaithful to the semantics of the underlying SPG. This is a good policy for a view to hold: it may not display everything, but what it displays is accurate. In some circumstances, this may not be the case, however, because the translator doesn't detect all possible topological errors. For example, the algorithm ensures that there is a one-to-one correspondence between a procedure's formal parameters and the initialization of those parameters, but it does not ensure that each parameter variable is initialized by data from the corresponding input port. Adding such checks is not conceptually difficult, but it is rather tedious, and it is hard to imagine how any but the most pathological of programs could yield an SPG containing the offending topology.

Application of my SPG-to-Pascal translator to the SPG of Figure 7-7 yields the program shown in Figure 7-10.[4] As one might hope, this generated program is identical to the original hand-written version in terms of its behavioral semantics. It is not textually identical, however, because the translator introduces some lexical noise in the form of unnecessary **begin-end** blocks and superfluous **else** clauses. Such artifacts are more annoying than misleading, however, and, as I noted earlier, it would require only a slightly more sophisticated translator to avoid generating them in the first place. For all practical purposes, my translator is able to faithfully recover the structure of the original Pascal program from the SPG that represents it.

Unfortunately, it is apparent from Figure 7-11 that my SPG-to-Pascal translator has considerably less success with the FSA-derived SPG of Figure 7-5. The translator performs well in identifying the high-level structural features of the program represented by the SPG, but it does substantially less well in coming up with Pascal depictions for the details of those features. The generated program accurately conveys the facts that there are three procedures in the program and that program execution consists of invoking the procedure

[4] Again, when examining Figures 7-10 and 7-11, bear in mind that none of the presentation-specific aspects of the program (e.g., location of line breaks and use of white space to show program structure) is produced by the translation algorithm.

```
program fsa(input, output);
  var c: integer;

  procedure error;
    begin
    write('Error!'); writeln;
    end;

  begin
  read(c);
  if (c < '0') or (c > '9') then begin error; end
  else begin end;

  repeat
    read(c);
  until (c < '0') or (c > '9');

  if (c <> '.') then begin error; end
  else begin end;

  read(c);
  if (c < '0') or (c > '9') then begin error; end
  else begin end;

  repeat
    read(c);
  until (c < '0') or (c > '9');

  if (c = 'e') or (c = 'E') then
    begin
    read(c);
    case c of
      '+': begin read(c); end;
      '-': begin read(c); end;
      otherwise begin end;
    end;

    if (c < '0') or (c > '9') then begin error; end
    else begin end;

    repeat
      read(c);
    until (c < '0') or (c > '9');
    end
  else begin end;

  write('Okay.'); writeln;
  end.
```

Figure 7-10: The Pascal program resulting from the Pascal-derived SPG of Figure 7-7.

```
program PASCAL001(input, output);

procedure DefaultFSAAction;
  begin
  end;

procedure DefaultFSAErrorRoutine;
  begin
  writeln('Error!');
  { I can't make any more sense of this, so I'm quitting here. }
  end;

procedure FSA001;
  begin
  { I can't make any more sense of this, so I'm quitting here. }
  end;

begin
FSA001;
end.
```

Figure 7-11: The Pascal program resulting from the FSA-derived SPG of Figure 7-5.

FSA001,[5] but it fails to provide any information on the actions that take place inside that procedure. This is a significant shortcoming, because the SPG subgraph corresponding to the contents of FSA001 is precisely that portion of the SPG that reflects the semantics of the original FSA.

It is easy to identify the feature of the SPG that cause trouble for the translation algorithm: it is the fact that there are two return nodes inside the Subprogram grouping for FSA001. In Pascal, a procedure must have exactly one exit location (the end of the procedure), and the translator recognizes that having two different return nodes is inconsistent with this restriction. There is no obvious way to approximate this in Pascal, especially not in the goto-less subset of the language I used for this experiment, so the algorithm simply refuses to try to translate the contents of the routine. It would be possible to reorganize the algorithm slightly so that the problem was ignored until the first return node in the routine was actually translated, but that would not be a general solution to the problem. It would also be possible to ignore return nodes when translating the contents of a routine into Pascal (leaving an appropriate comment where appropriate), but such an approximation strategy would yield a misleading view of the behavior of the program represented by the underlying SPG. There does not seem to be, in this case, any simple stratagem that allows the semantics of the SPG to be accurately represented in Pascal. In Section 6.4.1 I discussed various ways of dealing with such problems, but for the kind of "quick and dirty" SPG-to-view mappings I developed for this experiment, the approach I adopted in

[5]The names PASCAL001 and FSA001 that appear in the program are generated by the SPG-to-Pascal and the FSA-to-SPG translators, respectively. For details on the conditions under which such names are created, consult Appendix B.

my translator — that of simply refusing to continue with the subprogram — seems as reasonable as any. Certainly it is the easiest course of action, a consideration that is likely to make it a popular approach for user-defined views in practice.

Observations

I noted in my observations about the SPG-to-FSA translator that the mechanics of "parsing" the SPG — of moving over the graph to determine its topology — is tedious to program. That judgment was affirmed in my experience during the development of the SPG-to-Pascal translator. Possibly this difficulty is due to the fact that an SPG has a nonlinear form, possibly it is due to the fact that I have at my disposal no yacc-like parser generator for SPGs. Whatever the reason, this much is clear: when parsing an SPG, there are certainly a lot of error conditions to check for in an ad hoc manner. This problem seems to be a direct result of the generality of SPGs. Because so many SPG topologies are legal, and because so few semantic restrictions are placed on the content of an SPG, a robust SPG-to-view translator must be prepared to handle a wide variety of graphs as input. Further complicating matters is that, unlike standalone compilers for programming languages in conventional (loosely integrated) programming environments, a good SPG-to-view translator should try to provide *some* kind of reasonable output, regardless of the SPG it is attempting to translate. In this sense, SPG-based environments tend to encourage a COPE-style [7] attitude toward unanticipated input conditions: if an unexpected SPG topology is encountered, the translator should try to recover from the "error"and continue to generate a reasonable view. This is in sharp contrast to most compilers, which emit object code only if their input is entirely well-formed.

The presence of hyperedges in an SPG further complicates the process of parsing the graph. Their conceptual appeal is great, and I continue to believe that they express some relationships (e.g., alternative arms of a conditional statement) more directly than can be expressed using conventional graph edges, but there is no doubt that their great topological flexibility means that the tests involving them are tedious and involved. The fact that it is technically possible to have an edge with no branches adds to the complications. On the other hand, it is not obvious that replacing hyperedges with simple edge would simplify matters, because it is entirely possible that the same topological checks would be necessary, though in a different guise. This is in all likelihood another example of a general phenomenon: sometimes things that seem hard are just plain *hard*. Translating between arbitrary views is almost certainly one of these fundamentally difficult tasks, and for such tasks there is, as Fred Brooks has noted, no silver bullet [28].

A final observation is that SPGs allow more ambiguity about loop termination conditions than one might prefer. Consider this Pascal loop fragment:

```
for i := 10 to 20 do ...
```

In an SPG, the termination of this loop will be represented by an edge with two destination branches, one of which will have an annotation named Loop Exit. If we call the branch with this annotation B1 and we call the other branch B2, B1 might be controlled by any of the following conditions:

```
[i] > 20      20 < [i]      [i] >= 21      21 <= [i]
```

However, it is also possible for B1 to have the condition default, in which case B2 might have any of these conditions:

```
[i] < 10 or [i] > 20      10 > [i] or 20 < [i]
[i] <= 9 or [i] >= 21     9 >= [i] or 21 <= [i]
```

This is a large number of combinations to check for. A less syntactic and a more semantic representation of the constraints would simplify the process of determining the exit conditions, but then it would probably be more difficult to write an SPG-to-view translator that faithfully preserved the flavor of the condition's original concrete syntax. This may be another case in which the development of library routines to perform common operations may be the best approach to relieving the tedium of manually coding multiple tests.

7.6 Translating an SPG into a Runtime Stack

The Input

The mapping functions I developed work with any valid SPG.

The Approach

Any scope that is active should be displayed in a stack view, but in the context of an SPG, it is not necessarily obvious what it means for a scope to be active. I adopt the following conventions:

- All scopes are initially inactive.
- Scopes that are not also Subprogram or Rendezvous groupings become active when a token is placed on a port inside the scope. Such scopes become inactive when there are no tokens remaining on the subgraph in the scope. They also become inactive when an executable grouping containing them executes a Return node.
- Subprogram and Rendezvous groupings that are scopes become active when they are instantiated or invoked. They become inactive when they return. Because subprograms and rendezvous may be invoked non-strictly, it is possible for a subprogram or rendezvous scope to be active even if there are no tokens within it.
- When a stop node is executed, all active scopes remain active. This allows for a post-mortem examination of the state of the stack. All scopes become inactive when the SPG is prepared for execution, however.

Despite the name of the view, it is possible for multiple threads of control to be simultaneously active, hence for there to be more than one active stack. My algorithm has no trouble with this possibility, but it leaves the problem of how to actually display multiple stacks (or even a single stack) to PMs.

My algorithm assumes the following:

- The SPG is not modified by PMs while it is executing. If such modifications are allowed, the algorithm would have to be reworked to account for such on-the-fly modifications.

- A flow of control leaves a scope either by dying off at a dead-end (i.e., at a node with no output ports) or by leaving the scope as a token on an edge. There are pathological graphs that violate this assumption (i.e. a dead-end node that puts a token on an edge that leads nowhere), and it is possible to enhance the algorithm to detect such pathological conditions, but in practice it is probably not worth the trouble.

- If a scope S is active, so are all the scopes that S is inside of.

Because a runtime stack view is dynamic, the algorithm used to generate a stack view must be different in nature from the algorithms used to implement the static mappings of the previous sections. In particular, the algorithm must be primarily *reactive*, responding to events that take place within the SPG as the SPG executes. As such, my runtime stack view consists of several separate routines that respond to particular messages from the SPGM about events that are happening within the SPG.

The crux of my approach is to maintain in the view a data structure that keeps track of the active scopes in the SPG. When a scope becomes active, the variables declared in that scope are added to the view. When a scope becomes inactive, the variables it contains are removed from the view.

The view is designed to work correctly even if the SPG it is asked to display is in the middle of an execution when the view is created. As a result, the first thing the view does is inspect the SPG to determine which scopes are currently active. Given a set of active scopes, it is easy to inspect all Declaration nodes within each Scope grouping to identify all the variables located within the set of active scopes. The SPGM can be queried for the values of these variables, and the variable names and values can then be presented to users in the context of the scopes in which the variables are declared.

Once the set of currently active scopes has been identified, the view waits for notification by the SPGM that certain events have occurred. The impact of these events falls into one or both of the following categories:

- A scope should be added to or removed from the set of active scopes. Events in this category often correspond to the invocation of or the return from a subprogram instantiation.

- The value of a variable in an active scope should be updated. Events in this category typically correspond to the completion of a node's execution.

Detailed pseudocode for each event-handling routine in the Stack view can be found in Section B.5.

Observations

The great flexibility of SPGs means that a large number of arguably nonsensical topologies are legal. For example, it is possible to have edges leading nowhere, to have edges with some sources and/or destinations in a grouping and others outside the grouping, etc. Adding additional topological constraints to the definition of SPGs might well simplify the problem of writing SPG-to-view mappings, and if done carefully, would probably not impinge on their practical expressive power.

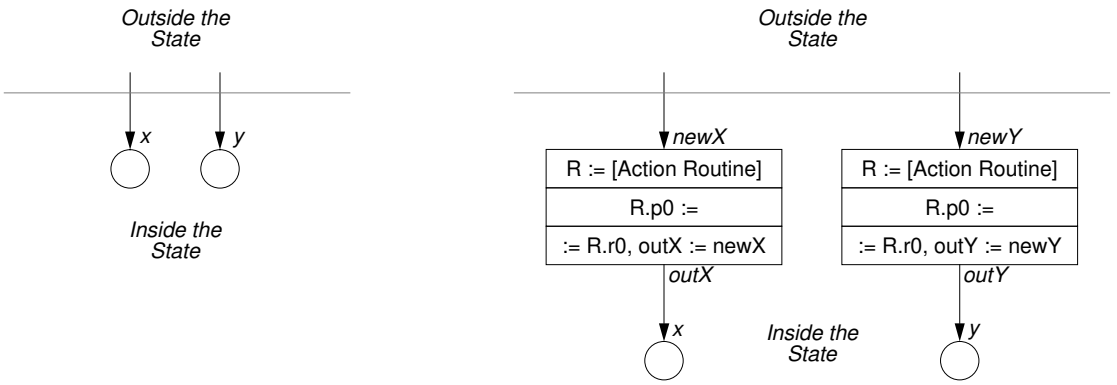


Figure 7-12: Left: two destination branches (parts of different FSA arcs) entering the SPG subgraph corresponding to an FSA state. Right: the same region of the SPG after addition of Call nodes so that the specified action routine is always invoked upon entry to the state.

7.7 Development of a Multiparadigm Program

For the purposes of developing a program using more than one view, it is convenient to start with the Pascal program of Figure 7-3 and then view it as an FSA, as shown in Figure 7-9. The SPG-to-FSA translator will determine that the action routine is undefined for each of the states in the automaton, a result that is unsurprising in view of the fact that the only procedure calls in the original Pascal program are to the routine responsible for signifying the presence of input errors.

To allow for the specification of an action routine for FSA states, it is necessary to have a routine in the FSA view that will make the necessary modifications to the SPG when an action routine is specified for a particular state. The design of such a routine is conceptually straightforward and is depicted in Figure 7-12. The underlying SPG must be modified so that any tokens entering the FSA state of interest via an FSA arc result in the specified routine being invoked before the tokens continue on to their original destination within the state. This can be accomplished by inserting Call nodes into the state so that all arcs entering the state terminate at a Call node instead of at their original destinations. Each Call node, in addition to invoking the appropriate action routine, forwards the token it receives from the arc on to the port at which the arc originally terminated; this ensures that the input to the remainder of the SPG representing the state is unchanged. A detailed pseudocode implementation for a routine that performs this transformation, resetActionRoutine, can be found in Section B.6.

If we assume that a routine newAction, shown in a Pascal view in Figure 7-13, has been added (most likely through a Pascal view, since there is no way to write to standard output through an FSA view) to the SPG so that newAction can be made to be the action routine for selected FSA states, and if we assume that the FSA routine resetActionRoutine has been invoked to set newAction to be the action routine for states A and E in the FSA, we end up with the SPG shown in Figure 7-14. If we then apply the SPG-to-Pascal translator to this SPG, we obtain the Pascal program shown in Figure 7-15. This new program is precisely what we would have obtained had we modified the software system through the Pascal

```
procedure newAction;
  begin
  write('This is a new action.');
  writeln;
  end;
```

Figure 7-13: Procedure newAction, to be used as an FSA action routine.

view directly. In this case, SPGs serve as an effective basis for multiparadigm software development.

Unfortunately, some examples do not work out this nicely. If we again view the modified SPG of Figure 7-14 as an FSA, and if we use resetActionRoutine to set the action routine for state B to newAction, we obtain a new updated SPG which, when passed through the SPG-to-Pascal translator, yields the incomplete program of Figure 7-16.

The cause of the incomplete translation is the updated topology of the SPG subgraph corresponding to FSA state B. This subgraph contains a loop, and when a call to newAction is added to the loop, the result is an exit-from-the-middle looping construct for which there is no corresponding Pascal control structure. Figure 7-17 shows the SPG corresponding to the modified loop. There is in principle no reason why a more sophisticated SPG-to-Pascal translator could not recognize loops such as these and translate them into Pascal equivalents, but the undertaking is unlikely to be trivial. A "natural" Pascal translation of this kind of loop is shown in Figure 7-18, and it involves the introduction of a new variable, the movement of the the loop termination test to a conditional construct, and creation of a new loop termination test. Whether view developers can be expected to perform transformations such as these is unknown; certainly they are unlikely to be provided in the kinds of "quick and dirty" views I employed in this experiment.

7.8 Conclusions

I stated at the outset of this chapter that my goal in performing the experiment described here was to arrive at a preliminary assessment of the strengths and weaknesses of SPGs in terms of the two primary constraints I introduced in Chapter 3: expressiveness and generalized invertibility. My experience with the development and application of FSA, Pascal, and Runtime Stack views provides me with a basis for the following preliminary conclusions.

First, there is much evidence to support the conclusion that the expressiveness criterion for SPGs has been met. I experienced no difficulties in mapping the semantics of Pascal or of FSAs into components of an SPG.

Second, my goal of imbuing SPGs with characteristics that would facilitate their being generally invertible appears to have been less successful. On the one hand, it was easy to create straightforward, unsophisticated mappings that translate an FSA into an SPG and then back again, and similarly for Pascal, so SPGs seem well-suited for mappings from a particular view to an SPG and then back again. The SPG-to-FSA translator also performed well with a Pascal-derived SPG, and I had no difficulty in writing a dynamic

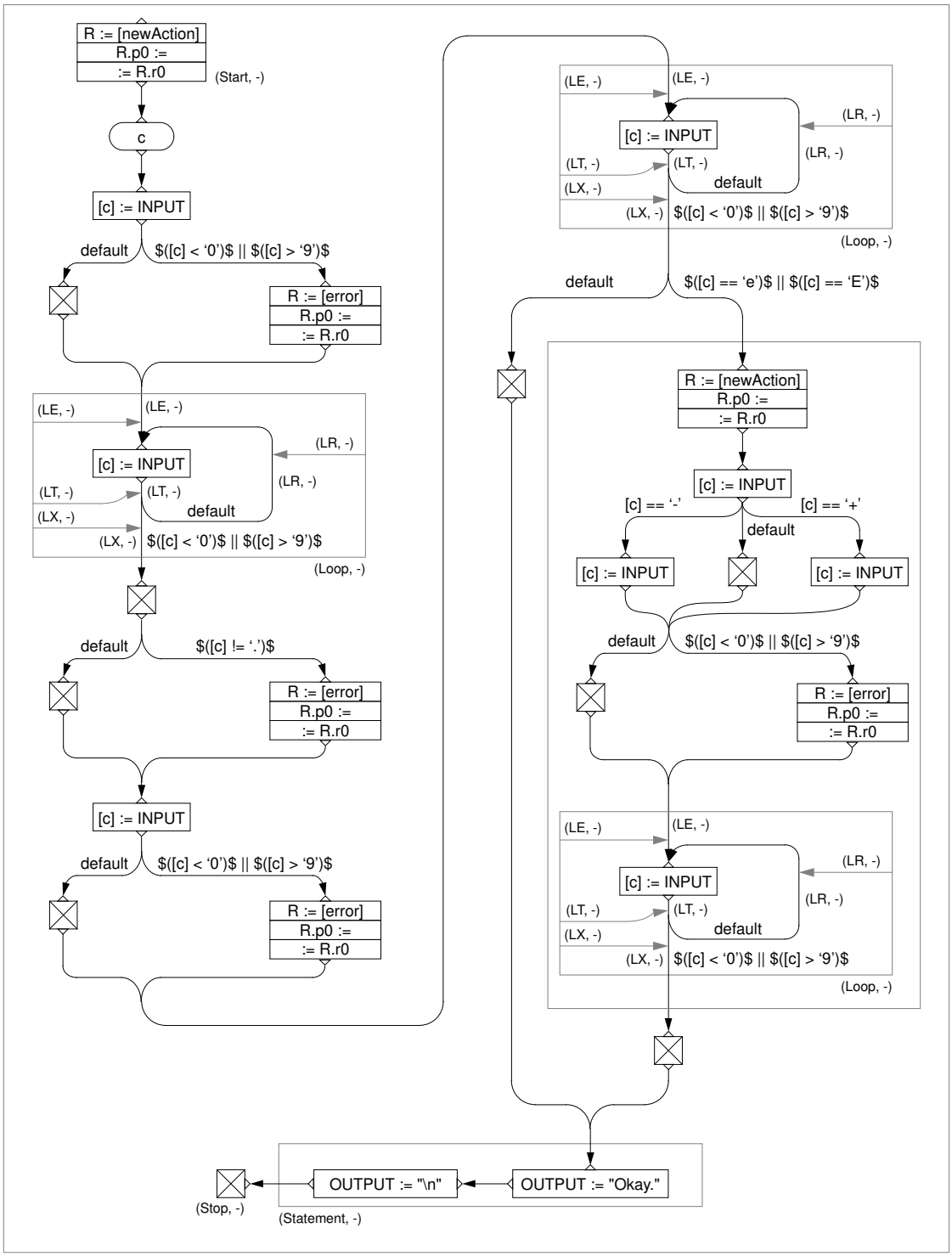


Figure 7-14: The SPG after modification to have FSA states A and E call newAction as their action routine. Loop annotations have been abbreviated as in Figure 7-7, and the subgraphs representing procedures Error and newAction have been omitted to save space.

```
program fsa(input, output);
  var c: integer;

  procedure newAction;
    begin
    write('This is a new action.'); writeln;
    end;

  begin
  newAction;
  read(c);
  if (c < '0') or (c > '9') then begin error; end
  else begin end;

  repeat
    read(c);
  until (c < '0') or (c > '9');

  if (c <> '.') then begin error; end
  else begin end;

  newAction;
  read(c);
  if (c < '0') or (c > '9') then begin error; end
  else begin end;

  repeat
    read(c);
  until (c < '0') or (c > '9');

  if (c = 'e') or (c = 'E') then
    begin
    read(c);
    case c of
      '+': begin read(c); end;
      '-': begin read(c); end;
      otherwise begin end;
    end;

    if (c < '0') or (c > '9') then begin error; end
    else begin end;

    repeat
      read(c);
    until (c < '0') or (c > '9');
    end
  else begin end;

  write('Okay.'); writeln;
  end.
```

Figure 7-15: The Pascal program resulting from the modified SPG of Figure 7-14. (Procedure error has been omitted to save space.)

```
program fsa(input, output);
  var c: integer;

  procedure error;
    begin
    write('Error!'); writeln;
    end;

  procedure newAction;
    begin
    write('This is a new action.'); writeln;
    end;

  begin
  newAction;
  read(c);
  if (c < '0') or (c > '9') then begin error; end
  else begin end;

  { I can't make any more sense of this, so I'm quitting here. }
  end.
```

Figure 7-16: The Pascal program that results if FSA state B also specifies newAction as its action routine.

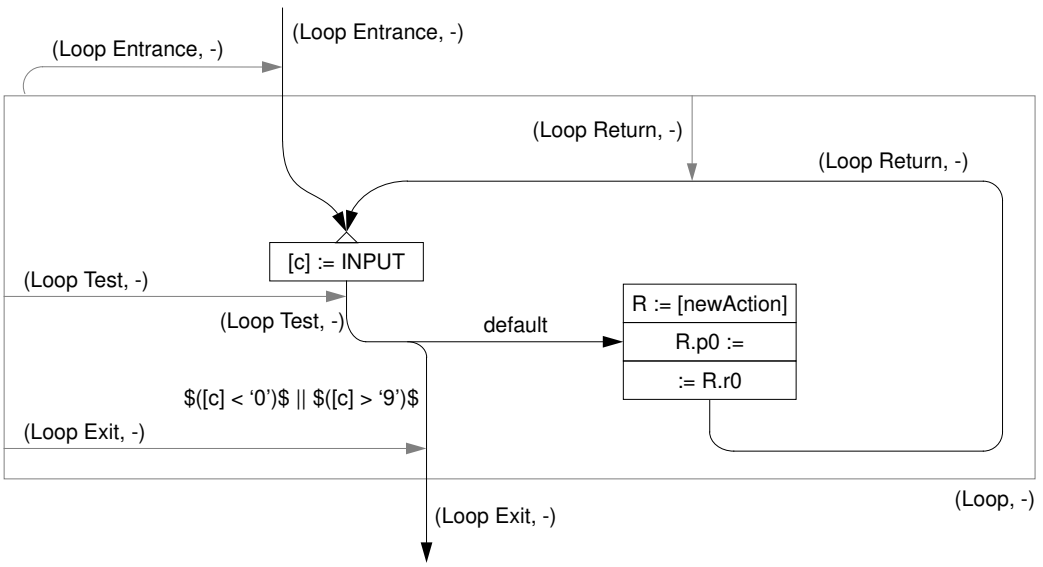


Figure 7-17: The SPG subgraph corresponding to the loop in state B after newAction is specified as the action routine.

```
repeat
  read(c);
  if (c < '0') or (c > '9')
    done := true;
  else
    newAction;
until done;
```

Figure 7-18: A Pascal approach to an exit-from-the-middle loop.

Runtime Stack view that works with any SPG. All these mappings lend credence to the thesis that the structure and semantics of SPGs are comprehensible enough to allow for their being mapped into a wide variety of views.

On the other hand, my SPG-to-Pascal translator produced results of limited utility when applied to the SPG generated from an FSA, and it showed itself to be less than robust in the face of small changes to the SPG originally derived from a Pascal program. It is possible that these results imply that SPGs are somehow fundamentally inadequate, but I believe a more justified conclusion is that the SPG-to-Pascal mapping I wrote was insufficiently sophisticated. Either way, it does seem to be a reasonable conclusion that effective multiple-view software development (including the kind of multiparadigm development I explored in Section 7.7) cannot be reliably based on simple, "quick and dirty" mappings that stop translating when they see an SPG construct for which there is no straightforward view-specific equivalent. It thus seems likely that the development of robust views for an SPG-based MVDE will be a nontrivial undertaking.

Further complicating the process of view creation is the nearly unbridled flexibility of SPGs themselves. One negative consequence of this great flexibility is that views must expend a fairly large amount of effort to make sense of the graph, a task whose difficulty is compounded by the fact that SPGs support both control-flow and dataflow. As a result, views must often search for more than one kind of topology when attempting to identify landmark features in the graph. In the SPG-to-view mappings I developed for this experiment, I employed ad hoc (and hence tedious) parsing techniques to perform these searches, but it may well be possible to take advantage of graph grammars to automatically generate SPG parsers that recognize meaningful topologies. Further reduction in the difficulty of implementing common view operations may be achieved through the development of libraries of routines that perform functions of use to a variety of views.

It may be possible to further simplify the task of generating SPG-to-view mappings by judiciously adding restrictions to the topology and content of valid SPGs. By further restricting the domain for translators from SPGs to views, it should be easier for such translators to perform their task. On the other hand, adding new constraints to the definition of an SPG runs the risk of making SPGs themselves more difficult for view developers to understand. Furthermore, adding restrictions on SPGs might complicate the activity of building SPGs in the first place — thus impinging on their expressiveness — and it might inadvertently restrict the range of as-yet-unimagined views. The primary reasons for my placing few restrictions on SPG topologies in the first place were to (1) facilitate the representation of incomplete programs and programs under construction and (2) allow

maximum flexibility for the development of future views. I believe that these two goals have been achieved, and if new restrictions on SPGs are to be imposed, one must be vigilant to ensure that the representational baby is not thrown out with the bath water. Nonetheless, I tentatively conclude that adding some topological constraints to the definition of SPGs would simplify the problem of writing SPG-to-view mappings, and, if done carefully, would probably not have any deleterious effects on their practical expressive power.

Chapter 8

Topics for Further Research

Ample testimony to the interest evinced by the research community in the topic of multiple-view software development can be found in the survey I provided in Chapter 2. However, virtually all of that work (the lone exception being the Garden system) is predicated on the assumption that a distinguished primary view is *the* source code, all other views in the environment being subsidiary to that view. Within these environments, subsidiary views — by definition derivations of the primary view — are almost never editable.

My research is fundamentally different from these previous efforts, because my interest is in MVDEs where there is no primary view and where a software system may be edited through any of a number of views in the environment. My analysis of existing integration mechanisms (Chapter 2), my view model (Chapter 3), my development of SPGs (Chapter 4), my examination of issues surrounding the use of SPGs in an MVDE (Chapter 6), and my analysis of the strengths and weaknesses of SPGs (Chapter 7) all help lay important groundwork for true multiple-view software development, but it hardly scratches the surface of the topic in its entirety.

I see opportunities for additional research in two basic areas. First, there is much that could be done with SPGs themselves. For example, SPGs contain a number of limitations and restrictions, and it would be interesting to know whether these limitations and restrictions can be relaxed or eliminated without reducing SPGs' suitability as a CR for an MVDE. Similarly, I suggested in Chapter 7 that SPGs might be easier to work with if they were *more* constrained, so an equally interesting line of work would be to try to restrict the legal topologies of SPGs without effectively reducing their expressiveness.

The second research area is more general, focusing on important issues in multiple-view software development that do not specifically concern SPGs. Such issues include the appropriate granularity of between-view change notification, the relationship between canonical program representations and database views, and the impact of a distributed development environment on the use of a canonical representation in the first place.

In the sections that follow, I sketch possible research topics in each of these areas.

8.1 Further Work on SPGs

The need for some enhancements to SPGs is fairly straightforward. For example, many real-world code-level views require better support for data types, the ability to distinguish

between declarations and definitions, and a richer set of mechanisms for passing parameters and results. Other potential improvements are less obvious, such as the need to allow for the expression of design restrictions within an SPG.

Types

The most glaring omission in the current specification of SPGs is its lack of support for data types. This limits the expressiveness of SPGs, because the only computations that can be represented are those dealing exclusively with integers. In addition, it precludes support for object-oriented languages (see Section 3.4), and it leads to the restriction that STask groupings must be statically allocated (see Section 4.3.3).

Adding type information to SPGs would be a nontrivial undertaking. In the same way that SPGs support both control-driven and data-driven execution, both strict and non-strict subprogram calls, and both serial and parallel execution, it would be highly desirable for SPGs to support both strong and weak typing. This would affect the definition of many executable components of the graph. For example, tokens, ports, and edges would all need to be augmented with type information. In addition, SPGs would require new features to allow for the definition of new types within a software system.

Adding support for types to SPGs would also complicate the business of creating views. Because the process for determining when a type error has occurred is highly view-dependent,[1] the responsibility for detecting type errors, like the responsibility for disambiguating references to identifier names (see Section 3.5.4), would fall to views. As in the case of common name-lookup algorithms, a library of type-checking algorithms could be made available for use by view writers; this would simplify the process of type checking for views employing common type-checking rules.

Subprogram Parameters and Return Values

SPGs currently support only pass-by-value for parameters and return values. It would certainly be interesting and worthwhile to add pass-by-reference, possibly by adding a new token type — reference tokens — to the formalism. Reference tokens would function much like pointers in most programming languages: the value of a reference token would be the VID of a variable containing the value of interest.

Declarations versus Definitions

Languages supporting the separate compilation of source files typically distinguish between the *declaration* of a name (a place where the name is introduced) and the *definition* of that name (a place where storage for the named object is allocated). Generally speaking, a name may have multiple declarations, but only a single definition.

SPGs currently fail to distinguish between declarations and definitions, which hampers their ability to model "what is going on" in languages like C and C++ that do draw a distinction. It would not be difficult to add support for this concept, however. All that is needed is a small set of new annotations and an expanded role for Declaration nodes.

[1] Example: most languages specify that an integer will be automatically converted to a floating point number if it is necessary to achieve type-consistency, but Ada performs no such implicit type conversions.

Declarations could be distinguished from definitions by standard annotations, and unexecutable groupings (or edges) could be used to identify equivalence classes of declarations, i.e., those declarations and definitions that correspond to the same named object.

Interprocess Communication

In Section 6.4.2 I described the mismatch between CSP's message-passing facility and SPG's STasks. The crux of the problem is that CSP insists on receiving messages from named callers, but STasks accept messages only from anonymous callers. It may be possible to unify these disparate approaches, however, in the same way that sequential subprogram calls were unified with interprocess entry calls, namely, by generalizing Accept nodes so that they *optionally* specify the executable grouping from which they will accept a message. Omitting such a specification would yield the current behavior (all calls to the appropriate entry would be accepted), but if a GID is specified for a caller, then only calls from that grouping would be accepted.

External Components

Other than reading from standard input and writing to standard output, there is currently no provision in SPGs for interacting with software components that are themselves outside an SPG. In particular, it is impossible for a software system represented as an SPG to call routines outside the SPG. For practical MVDEs, the ability to make such calls is probably essential.

Perhaps the most straightforward way to address this need is to create a new kind of grouping, the External Routine grouping. In the same way that an Entry grouping currently represents the interface to an STask entry, an External Routine grouping would represent the interface to a routine defined outside the SPG. Associated with the grouping — perhaps through an annotation — would be information on how to invoke the external routine. Calls to routines of this kind would take place through Call nodes, which would identify the routine to call by specifying the RID of the External Routine grouping representing the external routine.

Support for Restrictions

SPGs offer no way to represent *restrictions* on a software system. For example, it is not possible to express a design constraint stating that global variables must never exist. It is possible, of course, to employ a view disallowing global variables for the purposes of creating the initial SPG, but there no mechanism within an SPG that can prevent it from being edited by another view that is allowed to add a global scope. Another way of looking at this problem is to say that there is no way to distinguish "may not" from "does not" in an SPG. If an SPG has no global scope, that might simply mean that no view has yet found a need for one ("does not"), but it might also mean that no such scope is *supposed* to exist ("may not").

Adding support for restrictions in SPGs would increase their expressive power, but it would also run the risk of conflicting with the general policy of not representing view-specific constraints. It is important not to impinge on the integrity of this policy, because that policy is a cornerstone of SPGs ability to accommodate a wide range of views.

Form Canonicalization

One of my design criteria for SPGs was to generalize similar concepts in software development so that constructs that mean roughly the same thing in different views are represented in a uniform manner in an SPG. If this criterion were pushed to the limit, it would yield a *canonical form* for SPGs: a one-to-one mapping between concepts in software development and representations of those concepts in an SPG. Such a form would yield a number of benefits. Mapping from views to SPGs would be simplified by virtue of the fact that for any given view construct, there would be but a single way to represent it in an SPG. Similarly, mapping from SPGs to views would be straightforward, because each view would have no difficulty in interpreting portions of an SPG: each part could mean only one thing. Finally, the SPGM would able to assist views in offering high-level editing operations, because given an SPG in the canonical form and a change to be made to that SPG, the SPGM would often be able to deduce the additional changes required to restore the modified SPG to a canonical form.

The current specification of SPGs does not guarantee a canonical form for all programming concepts; views have enough latitude that they can represent similar concepts in different ways. For example, in Section 5.2.1 I discussed the difficulty of canonicalizing loops expressed as iterative constructs and loops expressed as tail recursion.

The problem of designing a canonical form is a difficult one. Work on the Plan Calculus, for example, has been in progress for over a decade, and there is still no canonical form for programs using that representation [173, p. 32–33]. That notwithstanding, it would be interesting to investigate whether some aspects of SPGs could be made to have canonical forms, because the dividends yielded by such forms are great.

Empirically Evaluating SPGs

The prototype implementation I developed for SPGs was sufficient to give me insights into the problems I was addressing, but it was not designed to act as a full-fledged MVDE for practical day-to-day use. Similarly, the paper experiment I performed yielded insights into the aspects of SPGs that affect the development of realistic views, but it is no substitute for true implementation experience with an SPG-centered MVDE. As a result, significant open questions remain as to whether an SPG-based environment will scale up to large programs and/or a large number of views, whether it will ultimately fulfill its promise of supporting as-yet-unwritten views that fit the view model, etc.

8.2 Related Research Areas

There is certainly a need for empirical work to assess the impact of multiple views on software development, and a number of questions that have traditionally been approached in a domain-independent fashion by researchers in the database community are particularly applicable to multiple-view development environments.

Evaluation of Multiple-View Software Development

It is still unknown whether multiple-view software development offers measurable advantages over more traditional program development techniques. The results of experiments

performed by Steven P. Reiss and me [137] suggest that increasing the number of views of a software system increases the performance of the programmers working on that system, but methodological shortcomings present in those experiments throw into question the data on which the conclusions are based. More empirical research is needed to validate the conclusions of that experiment, to characterize the nature of the interaction between software development tools, and to more carefully differentiate between issues of *using* a tool and issues of *learning* to use a tool.

Granularity of Change Notification

Given an MVDE, when should modifications to a software system made through one view be made visible to other views? As I noted in Section 6.1, a policy of immediate notification may be appropriate for some changes, yet inappropriate for others. This dichotomy implies the need for a flexible approach to change notification, one in which different granularities can be supported in an integrated, consistent fashion.

It may prove useful to adopt a database perspective when addressing this issue: a program representation can naturally be considered a database, and the interval between when a change is made and when that change becomes apparent to other views can naturally be considered a transaction. Therefore, it may be possible to adapt the research done on flexible transaction models [187, 13, 228, 143] to the particular needs of multiple-view software development environments.

Program Views and Database Views

The efforts expended by researchers interested in multiple views of programs and by researchers interested in multiple views of databases have generally been disjoint, Garlan's work on view-oriented databases being a notable exception (see Section 2.2.4). Work on updatable database views has usually focused on manipulating simple schemas, while developers of program views have been more concerned with designing appropriate schemas than with defining database views over those schemas. (This is the case with my work on SPGs, for example.) It would be interesting to cross-pollinate these fields, with the ultimate goal being the development of one or more schemas that are complex enough for practical use, yet still amenable to the kinds of formal manipulations that are necessary to ensure that consistency across views is always maintained.

Canonical Representations in a Distributed Environment

Most software development is carried out by teams of programmers, each member of which enjoys more or less exclusive use of a powerful personal computer or workstation with its own CPU, memory, and disk. Given this kind of distributed computing environment, it is not at all obvious that a monolithic CR for software systems is an appropriate foundation for an efficient, robust environment for software development [146]. However, it may be possible to employ techniques developed for distributed database systems [152, 153], whereby a CR has a logical structure that is monolithic, but a physical structure that is actually distributed. Use of such techniques can yield databases that are more efficient and more forgiving of network failures than are their physically monolithic counterparts, but further analysis is

required to determine whether these techniques will mesh well with the requirements of an MVDE.

Part III

Appendices and Bibliography

Appendix A

Mapping Between Petri Nets and Ada Tasks

In this appendix, I informally describe how restricted versions of Petri nets and Ada tasks can be mapped to and from SPGs, and I sketch how an SPG created through one view can be viewed through the other. I also examine some of the design choices facing implementers of views and presentations in an SPG-centered MVDE, and I discuss some of the difficulties that may be encountered when designing and implementing views.

A.1 Mapping Between Petri Nets and SPGs

The most straightforward mapping from a Petri net to an SPG is to map transitions in the net into empty nodes in the SPG and to map net places (including their input and output arcs[1]) into NP_1 control edges. This mapping, which is shown in Table A-1, faithfully reflects the structural features of a Petri net. It also yields the correct executable semantics, with the limitation that an SPG edge, unlike a Petri net place, may hold no more than one token.[2] Finally, the mapping is easily inverted: SPG nodes map to Petri net transitions, and NP_1 edges without branch conditions map to Petri net places. As usual, features present in an SPG and absent in a Petri net must be approximated, black-boxed, or omitted (see Section 6.4.1).

[1] I employ Peterson's terminology [155] in this appendix when I discuss Petri nets, hence transitions and places are connected by *arcs*, not edges. This has the advantage of differentiating between components of SPGs and Petri nets, because the former have edges, the latter, arcs.

[2] There is a recognized class of Petri nets that correspond to this restriction: *safe* (or *binary*) Petri nets [46]. Such nets never contain more than one token per place.

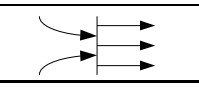
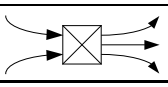
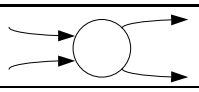
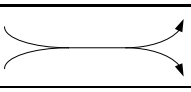
Petri net	SPG
	
	

Table A-1: A simple mapping from a Petri net to an SPG.

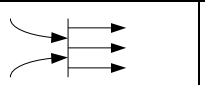
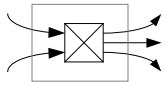
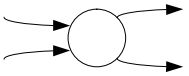
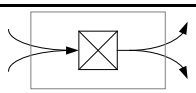
Petri net	SPG
	
	

Table A-2: A better mapping from a Petri net to an SPG.

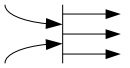
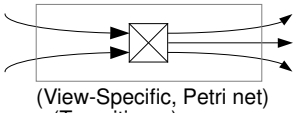
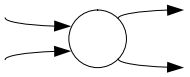
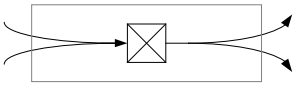
Petri net	SPG
	 (View-Specific, Petri net) (Transition, -)
	 (View-Specific, Petri net) (Place, -)

Table A-3: Adding annotations when mapping from a Petri net to an SPG.

This translation is not entirely satisfactory. The level of abstraction for Petri net places and transitions is typically much higher than is that for a single SPG edge or node, so mapping elements of a Petri net into single SPG components obscures the abstraction inherent in the use of Petri nets. In addition, Petri net transitions and places may themselves be replaced by more complicated subnets, but SPG nodes and edges are primitives. Both of these problems interfere with the goal of having an SPG accurately represent "what is going on" in a software system.

A better mapping is obtained by translating transitions and places in a Petri net into *regions* in the corresponding SPG; such regions are naturally delineated by groupings. This kind of mapping is depicted in Table A-2. This approach avoids the shortcomings of the previous mapping, but now there is a new problem: it is no longer obvious which features in an SPG corresponds to a transition or to a place in a Petri net. As a result, this mapping is more difficult to invert.

One way to eliminate the ambiguity is to have each SPG grouping generated by a Petri net view carry an appropriate annotation (see Table A-3), but that approach sidesteps the fundamental problem of MVDE software development: the challenge lies not in recovering a view from an SPG when you know the SPG came from that view in the first place, the challenge lies in coming up with ways to display an *arbitrary* SPG (possibly containing parts generated by different views) in terms of *arbitrary* views. View-specific annotations cannot help solve this problem.

It is certainly reasonable for views to add view-specific annotations to an SPG as a method of caching the results of earlier efforts to "make sense" of the graph, but any SPG-to-view algorithm that is *dependent* on the existence of such annotations is fundamentally inadequate. View-specific annotations are in this sense similar to compiler hints like C++'s inline directive. Just as a C++ compiler must accurately translate a C++ program, regardless of whether the program contains inline directives, so must a PM accurately translate an SPG

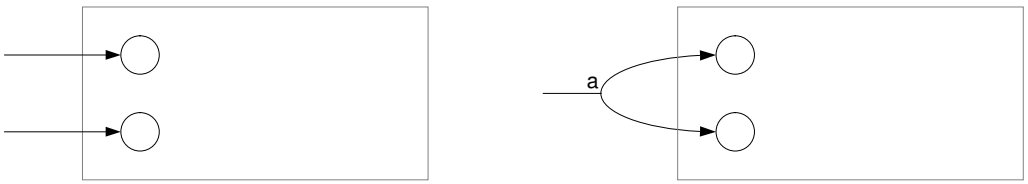


Figure A-1: Left: A grouping with executable semantics that are likely to differ from those of a Petri net transition. Right: A grouping with executable semantics consistent with those of a Petri net transition, even though branches entering the grouping lead to different nodes.

into a presentation, regardless of whether the SPG contains view-specific annotations. And just as a C++ compiler may be able to produce better code if inline directives are available, so may a PM be able to provide a more meaningful presentation if view-specific annotations are provided. In short, view-specific annotations may be reasonably employed for the purpose of *improving* the presentation of a view, but they should not be necessary to fundamentally *enable* the use of a view.

That being the case, the algorithm charged with the responsibility of making sense of an SPG as a Petri net must be designed under the conservative assumption that no view-specific annotations are present to guide the process. Such an algorithm could be based on the following heuristics:

- **If it is possible for execution in a grouping to commence before tokens have been placed on all branches leading into that grouping, the grouping cannot represent a transition.** Like an SPG node, a Petri net transition may not commence firing until it has a token on each of its inputs. A reasonable guideline is to assume that any grouping where input branches entering the grouping lead to different nodes cannot be a transition, because if different inputs to an SPG grouping lead to different nodes, it might be possible for one node in the grouping to be enabled for execution before the others are. This would violate Petri net semantics.[3]

  In practice, it is more useful to identify how an SPG feature *should* be translated into a view than to know how it should not be, because there are generally far fewer valid translations than invalid ones. In this case, it is better to invert the guideline above and follow this heuristic: if all branches leading into a grouping terminate at the same node, that grouping may correspond to a Petri net transition.

- **If it is possible for more than one token to leave a grouping for any single token entering the grouping, that grouping cannot be a place.** Petri net places are passive receptacles for tokens, not locations where new tokens are generated. If it is possible for more than one token to leave a grouping when only a single token enters, that grouping cannot represent a Petri net place. As before, it is more useful to develop guidelines that identify which groupings *could* be viewed as places, so

[3]This guideline is a heuristic, because it is more restrictive than is necessary. If two nodes in the same grouping both receive tokens from the same NP_a or $NP_{n>1}$ edge, the semantics of a Petri net transition are still satisfied. Examples of both of these situations can be found in Figure A-1.

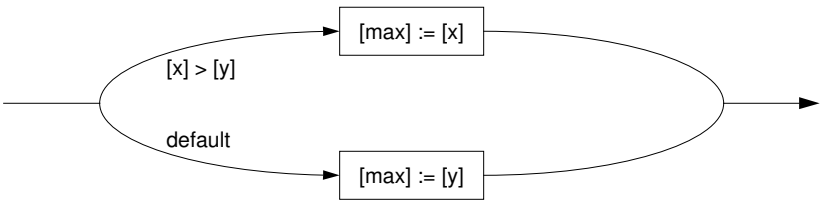


Figure A-2: An SPG for the Pascal statement, if x > y then max := x else max := y.

a reasonable first-order heuristic is that a grouping containing only NP_1 edges and containing only nodes with no more outputs than inputs may correspond to a place.

- **If the branch taken by a token leaving a grouping is nondeterministically chosen, the grouping must be a place.** Transitions must always place a token on each output arc, but places nondeterministically choose the output arc a token should follow. Hence, if an NP_1 edge with more than one destination branch leaves a grouping, that grouping must be a place.

  In this context, the concept of nondeterminism requires some examination. Nondeterminism in an SPG arises from edges with multiple destination branches, but it is important to note that not all edges of this kind allow truly nondeterministic token flow. If an edge's branch conditions are mutually exclusive, no dynamic nondeterminism will ever be encountered. Consider again the standard SPG structure for a conditional construct, originally introduced in Figure 4-25 and repeated here as Figure A-2. In this example, there are two destination branches, but one of them is controlled by the condition default. By definition, default is true iff the other branch conditions are false. These two branch conditions are mutually exclusive, so no nondeterministic runtime choice between them will ever be required. Conceptually, then, there is no nondeterminism present in the representation of a conditional construct.

  In the common case where multiple destination branches of an NP_1 edge are controlled by mutually exclusive conditions, one usually speaks of *conditional* execution, not *nondeterministic* execution. However, the relatively high level of abstraction of a Petri net may make it appropriate consider such execution to be nondeterministic. After all, a token might flow to the then part or to the else part; statically, the behavior cannot be determined. In fact, a Petri net view may wish to ignore some or all branch conditions entirely, treating all NP_1 edges with more than one destination branch as nondeterministic.[4] Such an approach would considerably simplify the task of generating a Petri net from an SPG, because branch conditions could be ignored.

- **Transitions can lead only to places, places can lead only to transitions, and everything is a place or a transition.** These are the fundamental restrictions governing the topological validity of Petri nets. Given an SPG grouping that is known

[4] This heuristic can be generalized. Given an NP_n edge with m destination branches, that edge can be considered nondeterministic whenever $m > n$. The Petri net translation of such an edge is a place with $\binom{m}{n}$ output arcs, each leading to a different transition with n outputs. Each transition represents a different combination of n of the m destination branches, and each output arc from such a transition represents one of the destination branches in the combination represented by the transition.

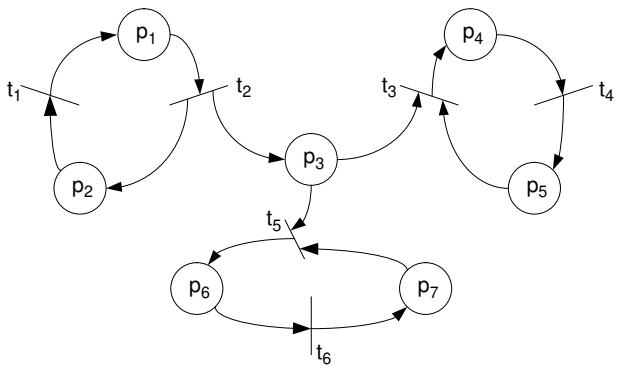


Figure A-3: A sample Petri net. This is essentially identical to Figure 1 of Peterson's survey [155].

to represent either a transition or a place, then, it becomes easy to determine whether groupings near the given grouping represent places or transitions.

Because everything must be a place or a transition, it is appropriate to adopt the inverse mapping of Table A-1 for nodes and edges in SPGs, i.e., to assume that nodes and edges correspond to transitions and places, respectively. However, an equally reasonable decision would be to ignore SPG nodes when mapping into a Petri net view, the justification being that individual SPG nodes are at too low a level of abstraction to be depicted in a Petri net view of a software system.

- **Nested groupings correspond to abstracted subnets.** This guideline follows from the fundamental assumption that SPG groupings correspond to Petri net transitions and places and from the fact that transitions and places may themselves be abstractions of more detailed subnets. It also implies that a reasonable strategy for making Petri net "sense" of an SPG is to perform a bottom-up analysis of the nested grouping structure.

- **Except for groupings, unexecutable components can be ignored.** A Petri net is a view of the executable aspects of a software system, so most unexecutable SPG components can be ignored when translating from an SPG to a Petri net. In general, unexecutable groupings cannot be ignored, however, because a transition or place might refer to some region of a program that does not correspond to a fundamentally executable structure. In fact, one can imagine views that display raw SPGs (i.e., in a form similar to the Gelo view of Figure 6-8) and allow users to specify subgraphs to enclose in unexecutable groupings corresponding to Petri net transitions or places. Such views could be used to add to these groupings the view-specific annotations of Table A-3.

Consider the Petri net shown in Figure A-3. A straightforward application of the mapping rules of Table A-2 yields the SPG in Figure A-4. To map this SPG back into a Petri net, it suffices to identify grouping p_3 as a place, not a transition. The structural feature of the SPG that signals this as the correct interpretation is the NP_1 edge with more than one destination branch leaving the grouping. This indicates the presence of nondeterministic token flow from the grouping, which can arise only in the form of a place in a Petri

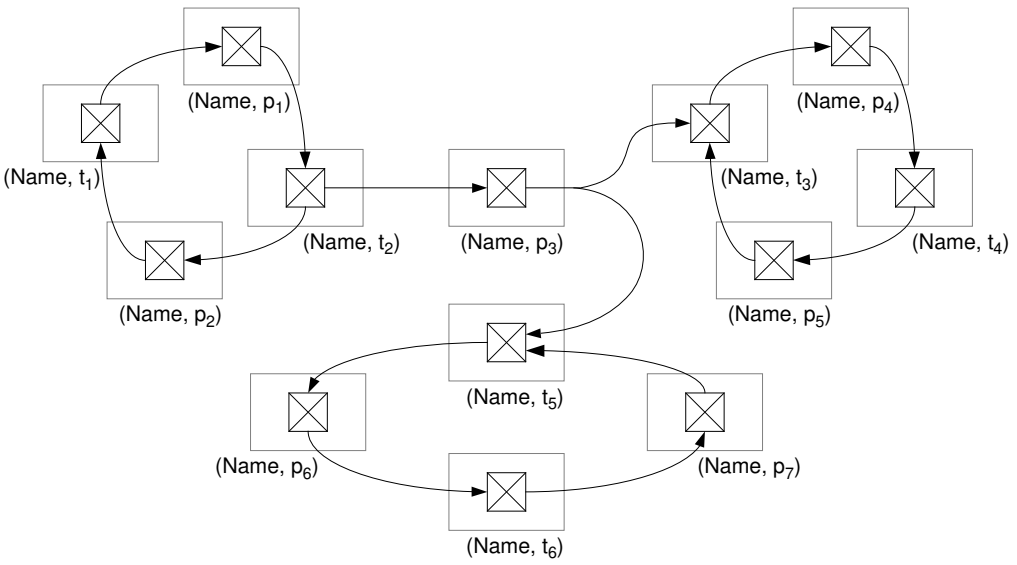


Figure A-4: An SPG corresponding to the Petri net of Figure A-3.

net. Once p_3 has been determined to be a place, the remaining groupings (each of which is consistent with the semantics of both a place and a transition) are quickly pigeonholed, domino-style, based on the rule that places and transitions must alternate. Of course, if each grouping contained a view-specific annotation indicating its role in a Petri net, the back-mapping from an SPG to a Petri net would be trivial.

A.2 Mapping Between Ada Tasks and SPGs

Figures A-5 and A-6 show an Ada program for an asynchronous, concurrent system consisting of one producer, two consumers, and a shared buffer. A high-level SPG for this program is shown in Figure A-7. The outermost grouping represents the global scope of the program, while the File grouping nested immediately within it represents the single file making up this example. In a real Ada program, each package would probably be contained within a separate file; that organization, too, would be easy to represent in an SPG. It is interesting to compare the File grouping of Figure A-7 with the File grouping of Figure 4-43 (the "hello, world" program), because the latter grouping demarcates a scope, while the former does not. This difference in representation reflects a difference in semantics between Ada and C. In C, a file comprises a scope, but in Ada it does not.

Each of the packages in the Ada program is represented as an unexecutable Scope grouping. This representation faithfully indicates the presence of an important structural unit in the program, but it fails to reflect the abstraction inherent in an Ada package (i.e., the fact that the internals of a package are inaccessible to code outside the package). Coming up with appropriate ways to represent such access restrictions is an interesting research question — see Section 8.1. In an open MVDE, it must be expected that one or more views will offer structural building-blocks that do not map directly to any of the SPG grouping types, and unexecutable groupings provide a flexible facility for indicating the presence of such structural features.

```
package Buffer_Package is
  task Buffer is
    entry insert(item: in Integer);
    entry remove(item: out Integer);
  end Buffer;
end Buffer_Package;

package body Buffer_Package is
  task body Buffer is
    count: Integer := 0;
    begin
      loop
        select when count < 10 =>
          accept insert(item: in Integer) do
            -- insert item into the buffer
            null;
          end insert;
          count := count + 1;
        or when count > 0 =>
          accept remove(item: out Integer) do
            -- remove an object from the buffer and assign it to item
            null;
          end remove;
          count := count - 1;
        end select;
      end loop;
    end Buffer;
end Buffer_Package;


package Producer_Package is
  task Producer;
end Producer_Package;

package body Producer_Package is
  task body Producer is
    n: integer;
    begin
      loop
        -- give n an appropriate value
        Buffer.insert(n);
      end loop;
  end Producer;
end Producer_Package;
```

Figure A-5: Ada code for implementing the buffer and producer tasks in an asynchronous, concurrent system consisting of one producer, two consumers, and a shared buffer. This code is modeled on that found in Ben-Ari's book [20]. In the interest of brevity, with and use directives have been omitted.

```
package Consumer1_Package is
  task Consumer1;
end Consumer1_Package;

package body Consumer1_Package is
  task body Consumer1 is
    n: integer;
    begin
      loop
        Buffer.remove(n);
        -- do something with n
      end loop;
  end Consumer1;
end Consumer1_Package;


package Consumer2_Package is
  task Consumer2;
end Consumer2_Package;

package body Consumer2_Package is
  task body Consumer2 is
    n: integer;
    begin
      loop
        Buffer.remove(n);
        -- do something with n
      end loop;
  end Consumer2;
end Consumer2_Package;


procedure producer_consumer is
begin
  null;
end producer_consumer;
```

Figure A-6: Ada code for implementing the consumer tasks and the main routine in an asynchronous, concurrent system consisting of one producer, two consumers, and a shared buffer. This code is modeled on that found in Ben-Ari's book [20]. In the interest of brevity, with and use directives have been omitted.

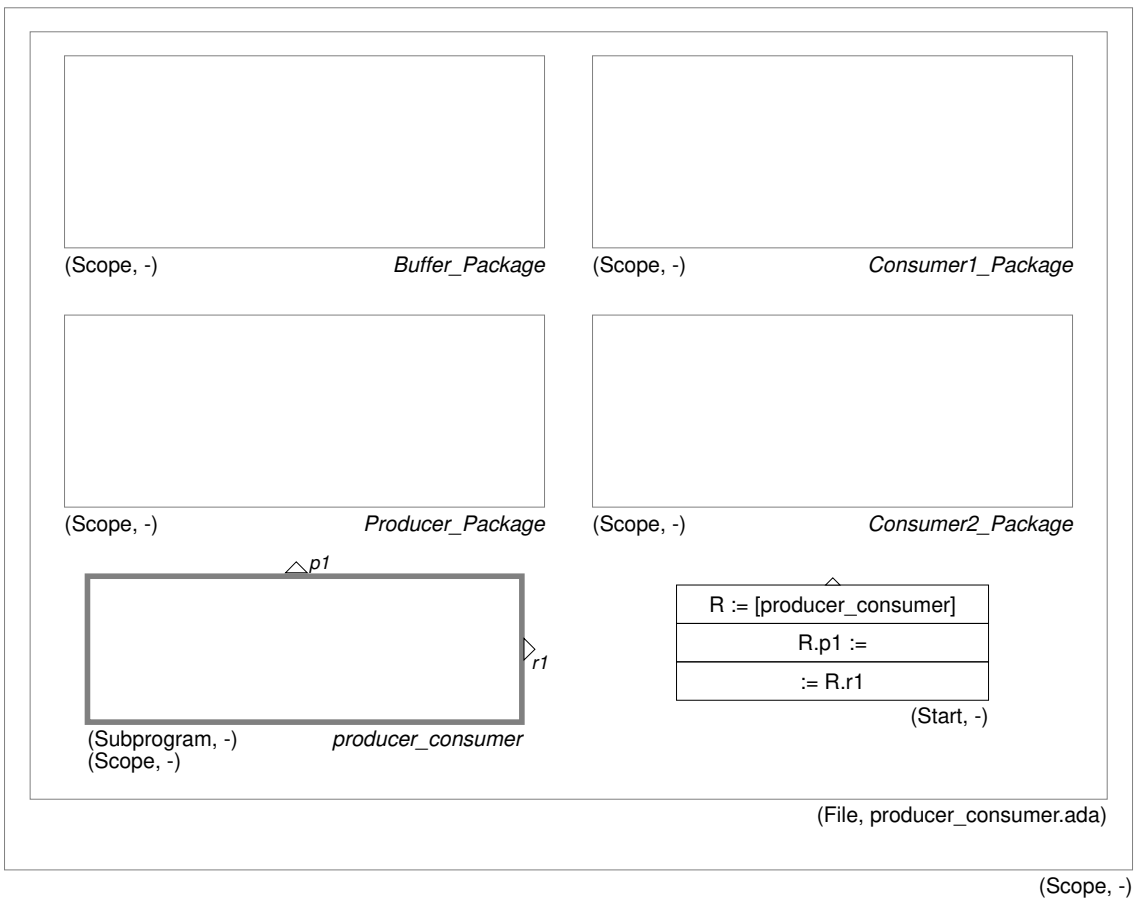


Figure A-7: A high-level SPG for the Ada producer-consumer program.

The presence of such undifferentiated groupings may or may not complicate the problem of mapping from an SPG to views. In this case, an Ada view can use a process of elimination to deduce that the four unexecutable Scope groupings inside the File grouping are best presented as Ada packages. The crux of the deductive process is the fact that each of the four groupings contains an STask grouping (see Figures A-9, A-10, and A-11), and an STask grouping always maps to an Ada task. Each grouping thus contains a task, and Ada tasks may only exist inside a subprogram, a block, a package or another task, so the unexecutable groupings must map to one of these structural features. However, they cannot represent subprograms or tasks, because such entities would be represented by Subprogram and STask groupings, respectively, nor can they correspond to a free-standing block, because blocks in Ada must themselves be contained inside other executable structures. The only remaining possibility is for the unexecutable groupings to correspond to packages.

The routine producer_consumer exists only to satisfy the Ada requirement that each program contain a main subprogram. As shown in Figure A-8, this routine returns immediately after being invoked. Similarly, the Call node that invokes producer_consumer exists only to invoke the main subprogram. Shortly after execution of the SPG is initiated, then, this Call node will have completed execution, and the only remaining active portions of the SPG will be those subgraphs contained in the unexecutable groupings representing packages.

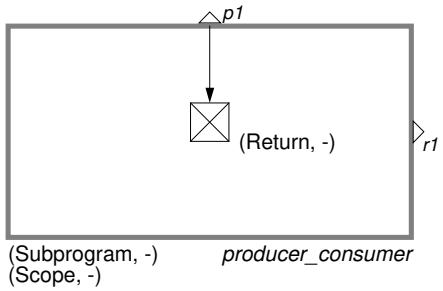


Figure A-8: An SPG for the producer_consumer routine.

The contents of these groupings are shown in Figures A-9, A-10, and A-11. The first two of these are not only straightforward translations of the features in the Ada source code, they are also structurally quite similar. The primary difference between them is the location of the comments in the loops. It would have been possible to associate the comments with the Call nodes (rather than with the Epsilon nodes), but that approach would make it more difficult for a view to discern that in the case of a producer, the comment precedes the entry call, but for consumers, it follows the call.[5]

The SPG for the buffer task was discussed in Section 5.2.4. The only difference is that here the STask grouping is shown in context, i.e., within an unexecutable grouping representing a package, while in Section 5.2.4, it was shown in isolation.

Like the Ada tasks they represent, none of the threads of execution in the SPG STasks of Figures A-9, A-10, or A-11 ever returns or terminates.

A.3 Viewing Ada Tasks as Petri Nets

One of the primary uses for Petri nets is to gain a high-level understanding of how concurrently and asynchronously executing agents may interact — typically communicate — with one another. These interactions may take many forms, but they are almost always based on one or more of the types of IPC I discussed in Section 3.5.5; even programs that communicate via shared memory rarely do so without some kind of concurrency control. All such IPC mechanisms are represented as STasks in SPGs, so it is possible to develop a general algorithm for mapping from SPGs to Petri nets by ignoring most of the graph and focusing exclusively on the STask groupings and the calls to the entries contained within them. The foundation of the algorithm is this simple observation:

- STasks synchronize when the Call node in the caller and the Accept node in the callee are both eligible for execution;
- Petri nets synchronize when each of the input places to a multiple-input transition contains tokens.

Each STask grouping can thus be viewed as a place in a Petri net, and each combination of a Call node and an Accept node that can handle that call can be viewed as a unique

[5] Another way to indicate which of the unexecutable (comment) node and the Call node in each figure should be presented first would be to use the Precedes annotation.

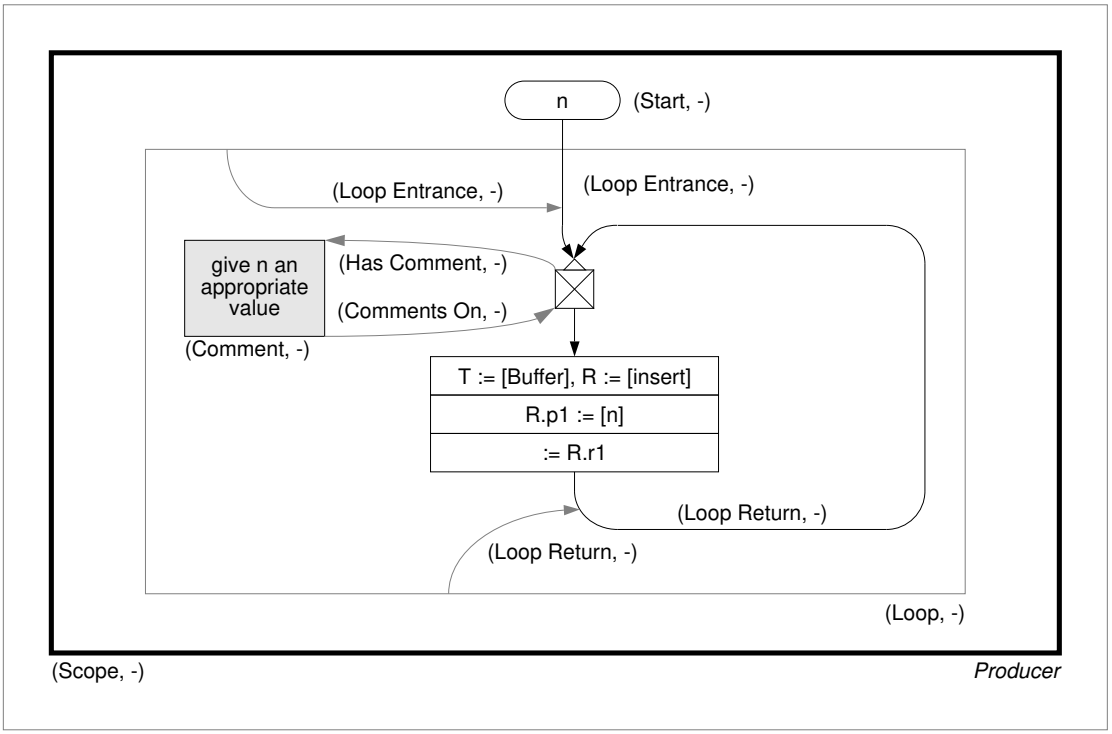


Figure A-9: An SPG for Producer_Package.

transition.[6] Interestingly, the SPG subgraph corresponding to the transition is easy to identify: it is the Rendezvous grouping executed at the behest of the Accept node. This correspondence is shown in Figure A-12.

Applying this simple algorithm to the SPG generated by the Ada producer-consumer program yields the Petri net of Figure A-13. Each STask in the SPG has become a place in the Petri net, and each of the Call/Accept node combinations has become a transition. A place has also been generated for the executable portion of the program not contained inside any task, i.e., for the subprogram producer_consumer. In general, the code not contained inside an Ada task executes in parallel with the code inside the tasks, and this code may contain entry calls, so it makes sense to treat all the non-task code as a "pseudo-task" (here called No Task) and to display it as a place in a Petri net view.

In this particular example, there is no interaction between the No Task code and the program's tasks, so it may seem appropriate for a view to elide No Task from the presentation. This is certainly a legitimate approach, but the presence of the No Task place may actually be useful when a static Petri net is modified to provide dynamic information about the execution of a system.

The most natural way to implement such a dynamic view is to show the movement of tokens through a static Petri net view. Unfortunately, the display of Figure A-13 contains too

[6] Branch conditions on Accept nodes can be ignored, the only drawback being that the resulting Petri net view may contain transitions corresponding to combinations of Call and Accept nodes that can, in practice, never occur.

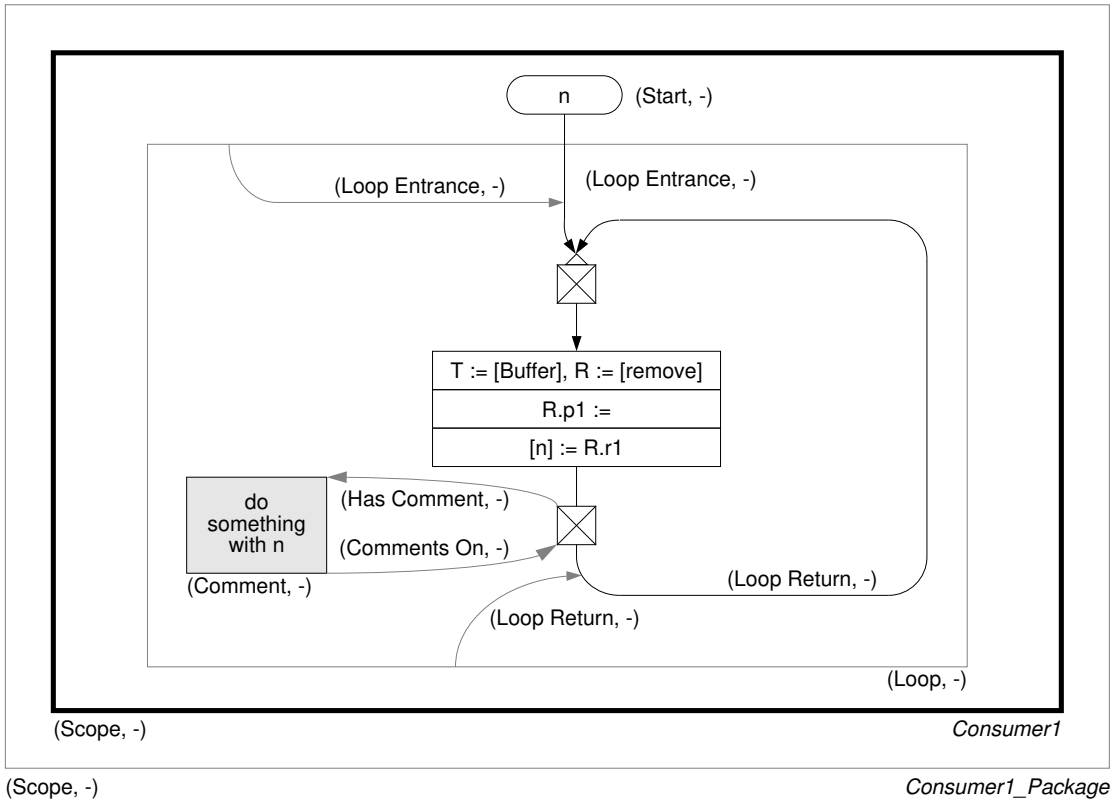


Figure A-10: An SPG for Consumer1_Package. The SPG for Consumer2_Package is essentially identical.

few places to yield any useful information — the level of abstraction is too high. However, it is a simple matter to increase the detail sufficiently to distinguish between two important kinds of task states: when a task is running independently and when it is awaiting synchronization with another task. This can be accomplished by dividing the SPG subgraph corresponding to a task (or to No Task) into two components: Call and Accept nodes, and *everything else*. Elements of the former component are depicted as unique places with arcs leading to and from synchronizing transitions, while all elements of the latter component are lumped together into a single place. Applying this more refined SPG-Petri net mapping to the producer-consumer example results in the Petri net of Figure A-14.

An interesting aspect of this second algorithm is that it maps STasks and their contents primarily into places; transitions arise only when a coupling exists between a Call and an Accept node. To indicate the flow of control among this plethora of places, the Petri net view introduces new transitions to enable the movement of tokens from places awaiting synchronization to places running independently. These transitions do not correspond to any part of the underlying SPG, but instead exist only to yield a Petri net that is topologically valid.

Because of its greater number of places, the kind of Petri net depicted in Figure A-14 is a much more suitable basis for dynamic views than is that shown in Figure A-13. The easiest way to provide such views is to show a Petri net token inside each place in the

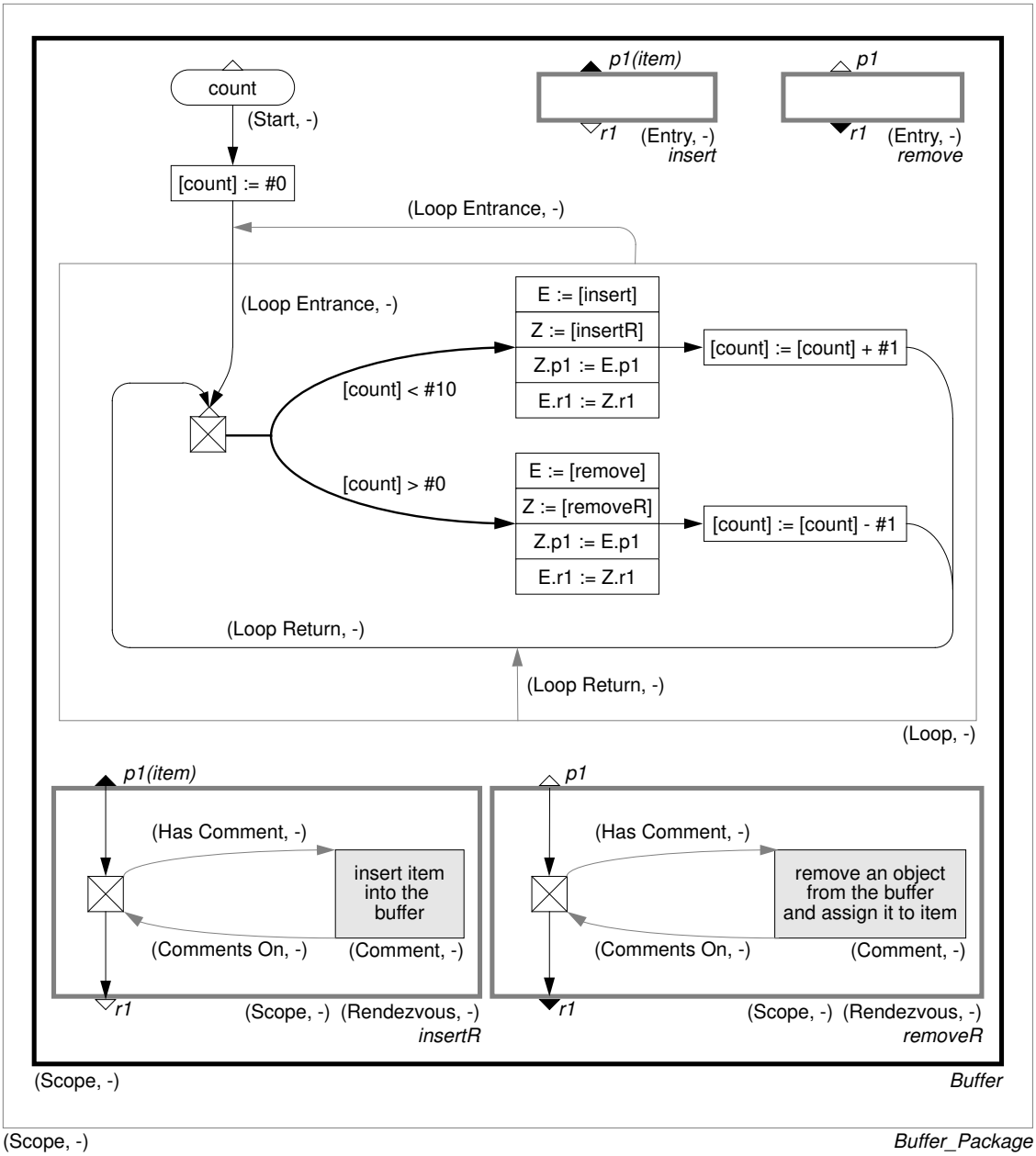


Figure A-11: An SPG for Buffer_Package.

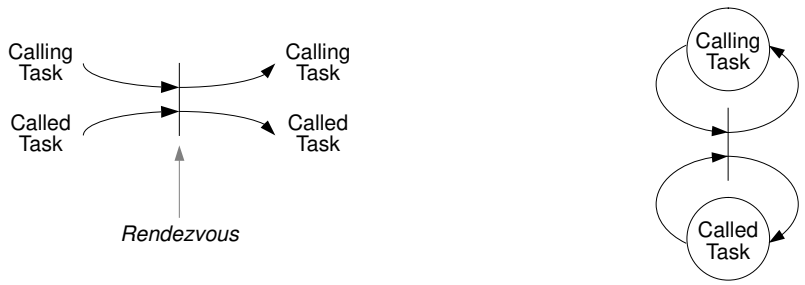


Figure A-12: Left: Representing rendezvous as a Petri net transition. Right: Representing STask groupings as places.

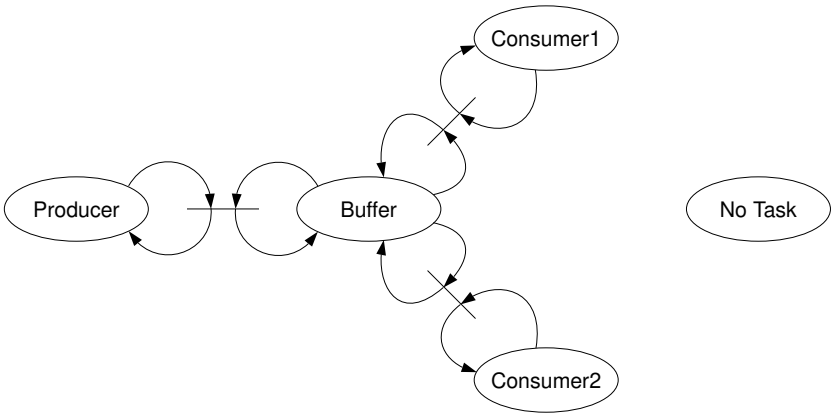


Figure A-13: A Petri net view of the producer-consumer program. The No Task place represents all the code in the program not contained inside any task.

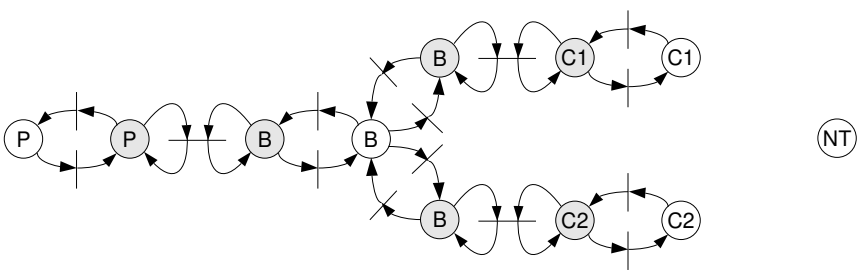


Figure A-14: A more detailed Petri net view of the producer-consumer program. States in which the program is waiting for synchronization are shown as shaded places. P = Producer, B = Buffer, C = Consumer, NT = No Task.

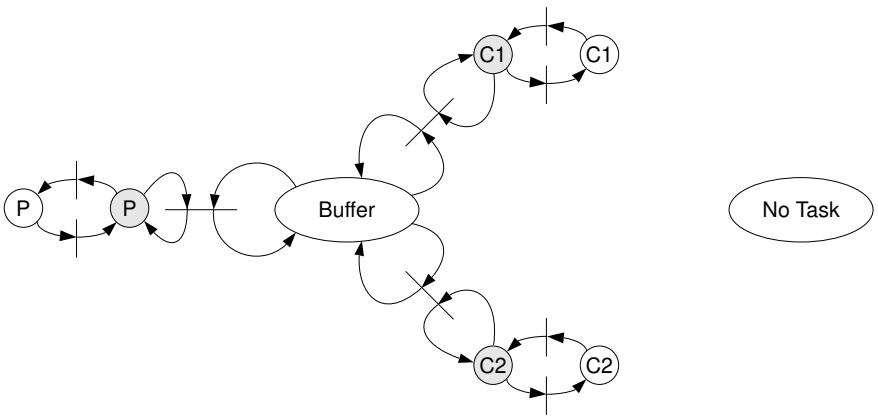


Figure A-15: A Petri net view of the producer-consumer program that combines the features of Figures A-13 and A-14.

presentation that has at least one SPG token in the underlying subgraph corresponding to that place. For example, when execution of the SPG initially begins, a token would appear in each Petri net place corresponding to a start node in the SPG. In the producer-consumer example, one such place would be No Task, but after execution of the producer_consumer subprogram had finished, the token would disappear. This behavior might be of interest to a developer working on the system, hence my comment earlier about not eliding No Task from the presentation.

Of course, it is possible to combine the algorithms used to yield Figures A-13 and A-14 so that a single presentation offers both capabilities. Such a view might allow a software developer to start with a view like Figure A-13, then selectively expand places until an image such as that shown in Figure A-15 had been produced. Further expansion of places and/or transitions could be based on the heuristics I presented in Section A.1.

Figure A-15 is in many ways similar to Figure A-3. This is far from coincidental — both are Petri net views of systems consisting of a producer, two consumers, and a shared buffer. Figure A-3 was produced by hand, however, while Figure A-15 could be generated automatically. That simple algorithms can be applied to SPGs to yield useful views such as Figure A-15 is important evidence that Semantic Program Graphs provide a solid foundation for extensible MVDEs.

Nonetheless, one might criticize Figure A-15 on the basis of the fact that it is not *identical* to Figure A-3, i.e., that the machine-generated Petri net fails to match the hand-drawn version exactly. Such criticism is unwarranted. The measure of success of an MVDE should not be whether it can generate output that is comparable to that of a human. Not even AI-based development environments are that ambitious [171]. Rather, it should be whether the MVDE is able to automatically produce *useful* views. The Petri net of Figure A-15 indisputably provides useful information — the independent, yet cooperative existences of the producer, the consumers, and the buffer are clearly delineated — and that is enough to deem its generation a success. It demonstrates that a software developer who creates a program using Ada can view that program as a Petri net by taking advantage of a tool that knows nothing about Ada, but much about SPGs.

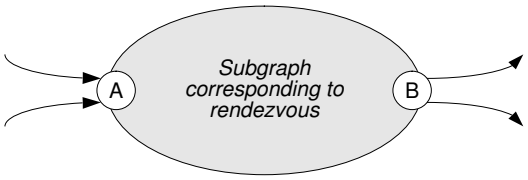


Figure A-16: An SPG subgraph (shaded) likely to correspond to the actions to be executed during a rendezvous. All tokens entering the subgraph come from node A, and all tokens leaving the subgraph are emitted by node B.

A.4 Viewing Petri Nets as Ada Tasks

Because of the close correspondence between tasking features in Ada and in SPGs, it is not difficult to translate an SPG containing STask groupings into an Ada view. The problem becomes more challenging, however, when the SPG contains no STask groupings to mark the boundaries of SPG subgraphs that are best viewed as Ada tasks. Nonetheless, such a translation may still be desirable, so it is possible to employ the following heuristics to identify features in an SPG that may indicate the presence of task-like interactions:

- **A node with two input ports corresponds to the onset of a rendezvous.** Such a node acts as a synchronization point for the two flows of control that place tokens on the ports. As a synchronization point, the node can be translated as either an `accept` statement (callee synchronization point) or an entry call (caller synchronization point) in an Ada program.

- **A node with two output ports corresponds to the completion of a rendezvous.** This kind of node transforms the single flow of control that comprises its execution into two separate flows of control, one beginning at each output port. It can thus be viewed as the endpoint of a rendezvous, i.e., as the termination of an Ada `accept` statement or entry call.[7]

- **The subgraph between a node marking the onset of a rendezvous and a node marking the completion of a rendezvous corresponds to the actions to be taken during a rendezvous.** A subgraph is "between" two nodes if the first node acts as a source for tokens flowing into the subgraph, and the second node acts as a sink for tokens leaving the region. This idea is shown in Figure A-16. The two nodes marking the onset and completion of rendezvous are likely to be best viewed as an `accept` statement, because they are topologically adjacent to the subgraph corresponding to the rendezvous, and, in an Ada program, the actions to be taken during rendezvous are specified in the called task (i.e., in the task `accept`ing the call), not in the calling task.

[7] Of course, a node with two output ports can also be viewed as a node that has nothing to do with a rendezvous, but instead spawns a new flow of control. The fact that two different interpretations are possible should not be surprising; the notion of interpreting a node with two output ports as the end of a rendezvous is only a heuristic. The goal of an Ada view is to make sense of an SPG in any way it can, so it might even employ one interpretation in one place and a second interpretation in another. In general, there is no "right" way to present an SPG in terms of a particular kind of view, there are only better and worse ways to do it.

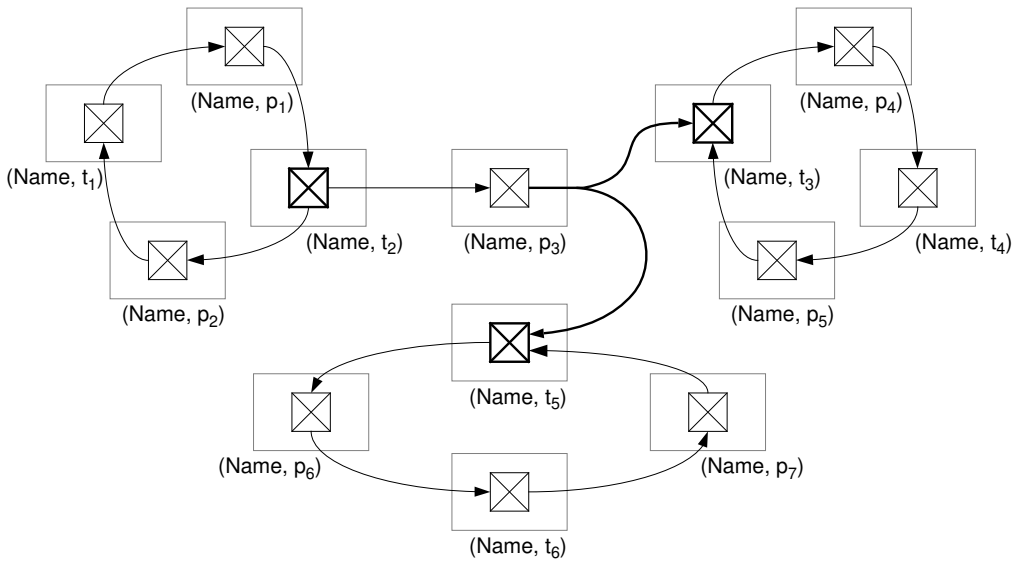


Figure A-17: Nodes and edges in the SPG of Figure A-4 that trigger heuristics for mapping from SPGs to Ada tasks.

- **An NP edge allowing nondeterministic token flow corresponds to a `select` statement.** The only Ada construct that exhibits nondeterministic behavior is the `select` statement, so any truly nondeterministic choice in an SPG (see Section A.1) must be translated into a `select` statement.

Application of these heuristics to the SPG derived from the Petri net of Figure A-3 identifies three nodes and one edge that are likely to be components of tasking interactions. These SPG components are highlighted in Figure A-17. Given these components, the next problems to tackle are those of determining how many tasks are present and which SPG subgraphs make up the contents of each identified task.

A reasonable approach to solving these problems is to examine the topology of an SPG around rendezvous sites, because such sites correspond to locations where disjoint flows of control merge (at the onset of rendezvous) and then later break apart (at the completion of rendezvous). By identifying the subgraphs making up the independent flows of control participating in each rendezvous, the SPG can be partitioned into subgraphs that must be assigned to separate Ada tasks. These subgraphs will not quite be disjoint, because nodes representing the onset and completion of rendezvous correspond to both call and `accept` statements in Ada. Such nodes will therefore map to two Ada tasks, not one. A presentation of the SPG as a collection of Ada tasks can then be generated that is consistent with the partitioning of the subgraphs.

Unfortunately, it is not a simple matter to determine which of the two flows of control leaving a rendezvous corresponds to either of the two flows of control entering that rendezvous. Consider the leftmost diagram of Figure A-18. Nodes `w` and `x` (and the subgraphs leading to them) are clearly parts of separate flows of control, one corresponding to the calling task and one corresponding to the called task. Hence they should be assigned to separate tasks in an Ada view of their SPG. Similarly, nodes `y` and `z` (and the subgraphs emanating from them) must also be assigned to separate tasks. Finally, the task to which

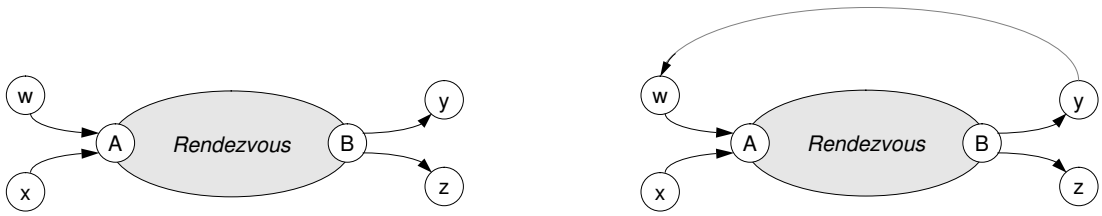


Figure A-18: Ambiguity in determining task assignments. Left: Does w belong to the same task as y or as z? Right: A path from y to w (only) indicates that they are part of the same task and that x and z are part of a separate task.

w is assigned must be the same as the task to which either y or z is assigned (and similarly for x, y, and z), but how is it to be determined whether w belongs with y or with z?

In general, it may not be possible to make such a determination, but if there is a path through the executable portion of the SPG from a flow of control leaving a rendezvous to exactly one of the flows of control entering that rendezvous, the subgraph corresponding to those flows of control must be assigned to the same Ada task. This heuristic is shown in the rightmost diagram of Figure A-18, where the existence of a path from y to w establishes that they belong in the same task. It also indirectly establishes that x and z belong in the same task, but one that is separate from the task containing w and y.[8]

The results of applying this heuristic to the SPG of Figure A-17 is shown in Figure A-19. In this example, the nodes in the groupings t_2, t_3, and t_5 "look like" the end, the beginning, and the beginning of rendezvous regions, respectively, but traversal of the graph yields no corresponding beginning, end, or end nodes, respectively. A view might reasonably choose to black box such incomplete rendezvous sites, but in this case I have adopted the convention that if no complimentary delimiter can be found for a node seeming to mark the beginning or ending of a rendezvous region, that node will be assumed to comprise the entire region. It then becomes easy to establish the following:

- Nodes in the t_2-p_2-t_1-p_1 cycle must be in the same Ada task and, except for t_2, must be in a different task from the subgraph reachable from p_3.
- Nodes in the t_3-p_4-t_4-p_5 cycle must be in the same Ada task and, except for t_3, must be in a different task from the subgraph reachable from p_3.
- Nodes in the t_5-p_6-t_6-p_7 cycle must be in the same Ada task and, except for t_5, must be in a different task from the subgraph reachable from p_3.

The only task assignment consistent with these restrictions is a separate task for each of the three cycles, with the node in the grouping p_3 being outside each of those tasks. There is no *a priori* reason to place p_3 in its own task rather than in the non-task portion of an Ada program, but the "nondeterministic-looking" edge leaving p_3 maps to an Ada select statement (by an earlier heuristic), and Ada requires that such statements occur only inside a task body.

[8] If it is not possible to unscramble the flows of control entering and exiting a putative rendezvous, it becomes more difficult to identify tasks and their interaction sites, hence to generate an intelligible presentation of the SPG. In the worst case, black boxes must be employed. However, even a view that successfully generates a good presentation only some of the time may well be useful to software developers.

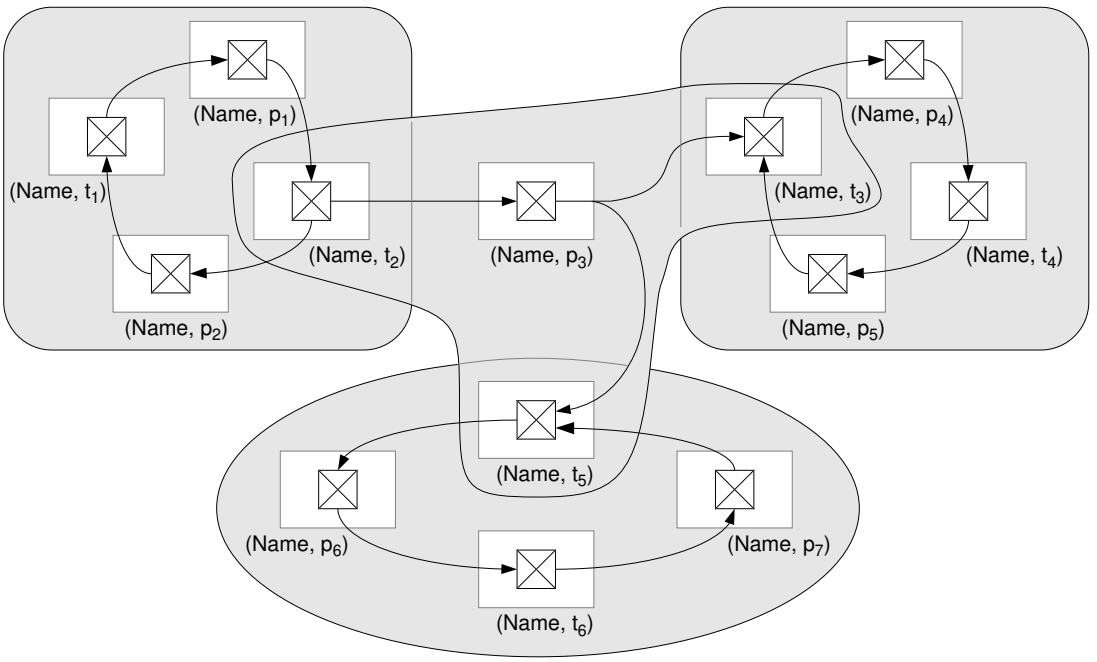


Figure A-19: Task divisions in the SPG of Figure A-4. Nodes in the groupings t_2, t_3, and t_5 are each part of two tasks because of their roles as rendezvous sites.

An Ada view must provide a name for each of the tasks it puts in a presentation, and, in this case, there is no obvious way to derive task names from the SPG. One might reasonably expect an Ada view to generate generic names like Task1, Task2, etc., but for purposes of exposition here, I use the task names T2 (for the task containing the t_2-p_2-t_1-p_1 cycle), T3 (for the task containing the t_3-p_4-t_4-p_5 cycle), T5 (for the task containing the t_5-p_6-t_6-p_7 cycle), and P3 (for the task containing p_3).

Having determined the number and content of each of the tasks to be presented in the Ada view of this SPG, the primary remaining difficulty is that of determining, for each rendezvous site, which task is the caller and which is the callee. The P3-T3 rendezvous and the P3-T5 rendezvous are easy to analyze, because the edge leading from P3 to these two rendezvous sites has already been determined to correspond to a select statement, and a select statement can lead only to an accept statement. Hence, T3 and T5 must be callers, and P3 must be the callee. The P3-T2 rendezvous is more troublesome, because the topology of the SPG is consistent with both possibilities: both P3 and T2 can be viewed as either caller or callee. There being no way to resolve this ambiguity based on the structure of the SPG, the best approach is probably to black-box the call in both tasks.

Given task boundaries, task contents, and a characterization of the caller and callee at each task interaction site, it is possible to present an Ada view of the SPG generated from the Petri net of Figure A-3. Such a view can be found in figures A-20 and A-21. Because Ada does not allow free-standing tasks at global scope, the view has added packages to surround each task. The view has also generated names (entry1 and entry2) for the entries called at the sites of the P3-T3 and the P3-T5 interactions. There is no main program in this view, but there are no subprogram groupings in the underlying SPG, either. This is

```
package P3_Package is
  task P3 is
    entry entry1;
    entry entry2;
  end P3;
end P3_Package;

package body P3_Package is
  task body P3 is
    begin
      -- ??? Call T2 or accept call from T2 ???
      select
        accept entry1 do
          null;
        end entry1;
      or
        accept entry2 do
          null;
        end entry2;
      end select;
    end P3;
end P3_Package;


package T2_Package is
  task T2;
end T2_Package;

package body T2_Package is
  task body T2 is
    begin
      loop
        null;
        -- ??? Call P3 or accept call from P3 ???
        null;
        null;
      end loop;
  end T2;
end T2_Package;
```

Figure A-20: Partial Ada view (tasks P3 and T2) of the Petri net of Figure A-3. In the interest of brevity, with and use directives have been omitted.

```
package T3_Package is
  task T3;
end T3_Package;

package body T3_Package is
  task body T3 is
    begin
      loop
        null;
        P3.entry1;
        null;
        null;
      end loop;
  end T3;
end T3_Package;


package T5_Package is
  task T5;
end T5_Package;

package body T5_Package is
  task body T5 is
    begin
      loop
        null;
        P3.entry2;
        null;
        null;
      end loop;
  end T5;
end T5_Package;
```

Figure A-21: Partial Ada view (tasks T3 and T5) of the Petri net of Figure A-3. In the interest of brevity, with and use directives have been omitted.

not a problem, because an SPG need not represent an entire program.

More significant is the lack of start nodes in the SPG. The ramifications of this omission for dynamic views are less serious than might be supposed, because tokens can be manually placed on an SPG by making appropriate calls to the SPGM (see Section 6.2.1). However, the fact that there are no start nodes in the tasks T2, T3, and T5, in conjunction with the fact that the SPG subgraph in each task forms a cycle, makes it impossible to determine an ordering for the statements making up the loop in the Ada view of each of the cycles. In Figures A-20 and A-21, a statement ordering for the contents of these loops has been chosen arbitrarily, but this is unlikely to be a satisfactory general solution to the problem.[9] Fortunately, it is unlikely that, in practice, SPGs will often contain cycles without entry points.

This Ada program has a structure that is remarkably faithful to "what is going on" in the original Petri net. The primary omission is the looping control structure one expects to be present in task P3, i.e., the task corresponding to the buffer. The body of task P3 in Figure A-20 consists of a black-boxed interaction with T2 (indicated by a comment) followed by a single nondeterministically selected rendezvous with a caller of either entry1 or entry2. Upon completion of the rendezvous, P3 ceases execution. Clearly, this is not the behavior one expects from a buffer in a producer-consumer system.

Examination of the original Petri net (Figure A-3), however, reveals that this looping structure is not present in the buffer portion of the net, either, even though it is present for the portions of the net corresponding to the producer and the two consumers. In fact, it is apparent that the original (human-generated) Petri net is actually modeling *two different aspects of the system at the same time*. Within the t_2-p_2-t_1-p_1 cycle (the producer), the t_3-p_4-t_4-p_5 cycle (a consumer), and the t_5-p_6-t_6-p_7 cycle (the other consumer), tokens represent *control flow*, but for arcs leading to and from p_3 (the buffer), tokens represent *dataflow*. When translating from SPGs to Ada, however, the view uniformly interpreted executable edges as indicating control flow.

This kind of inconsistency in modeling is difficult to detect algorithmically, especially for abstract views like Petri nets, but it is probably not uncommon for systems produced by real developers. That being the case, it is encouraging that it is possible to develop a mapping from SPGs to Ada that yields meaningful results even in the presence of such underlying inconsistencies. The existence of such mappings is important evidence that SPGs can serve as a reasonable foundation on which to build multiple-view development environments.

[9] In this example, the information in the SPG so underconstrains an Ada view that it might be advisable to prohibit editing through that view. Other SPGs, of course, might not be so underconstrained. This leads to the possibility that a view might dynamically decide whether to be read-write or read-only depending on the input it has to work with.

Appendix B

Pseudocode for Sample Mappings

In this appendix I provide pseudocode for the mappings I discussed in Chapter 7. I employ a generic Algol-like syntax within the pseudocode, I use C++'s `//` symbol to indicate the beginning of a comment-to-end-of-line, and I also occasionally use C++'s notation for default parameter values. I assume short-circuit evaluation of logical expressions (as in C), and I indicate the extent of compound commands through indentation rather than through the use of explicit `begin...end` delimiters.

The pseudocode programs in this appendix were designed only to demonstrate that the various mappings could be implemented without undue effort. As "quick and dirty" programs, they were not designed with efficiency in mind. They can therefore be expected to be both more comprehensible and less efficient than would be the more performance-sensitive implementations that one would be likely to employ in practice.

The entry point of each program that follows is at the beginning of the code. Procedures and functions that are used in a mapping follow the end of the "main block."

B.1 Translating an FSA into an SPG

The program below assumes that auxiliary data structures exist to store the following information for each state: its name (possibly null); whether it is a start state; whether it is a final state; and the name of the routine to call when the state is entered (also possibly null). Similarly, the program presupposes the existence of a local data structure that stores the following information for each arc: a way of identifying the state at which it originates; a way of identifying the state at which it terminates; and a description of the inputs that enable traversal of the arc.

```
// If there is no default action routine in the FSA, create a routine for
// the purpose and add it to the SPG.
If (no default action routine has been specified)
  Create a new Subprogram grouping G;
  Give G an annotation ("View-Specific", "FSA") and give this annotation
    an annotation ("Default Action", "");
  Give G an annotation ("Name", "DefaultFSAAction");
  Give G an annotation ("Scope", "");
  Give G a single control InOut port with annotation ("PR Name", "p0");
  Give G a single control OutIn port with annotation ("PR Name", "r0");
  Create a new Epsilon node N and add N to G;
  Give N a single control input port P;
  Give N an annotation ("Return", "");
  Add to G a control NP1 edge from p0 to P -- put no condition on the
    edge's destination branch;
  Add G and everything in/on G to the SPG;

// If there is no default error-handling routine in the FSA, create one.
If (no default error-handling routine has been specified)
  Create a new Subprogram grouping G;
  Give G an annotation ("View-Specific", "FSA") and give this annotation
    an annotation ("Error Routine", "");
  Give G an annotation ("Name", "DefaultFSAErrorRoutine");
  Give G an annotation ("Scope", "");
  Give G a single control InOut port with annotation ("PR Name", "p0");
  Give G a single control OutIn port with annotation ("PR Name", "r0");
  Create a new Multiassignment node N containing
      OUTPUT := "Error!\n"
    and add N to G;
  Give N a single control input port P;
  Give N an annotation ("Stop", "");
  Add to G a control NP1 edge from p0 to P -- put no condition on the
    edge's destination branch;
  Add G and everything in/on G to the SPG;

// Translate each connected component of the FSA into a subprogram.
For each connected component CC of the FSA:
  Create a new Subprogram grouping G;
  If (CC has a name)
    Give G an annotation ("Name", name of CC);
```

```
Else
  Generate a unique name N for G and give G an annotation ("Name", N);
Give G an annotation ("Scope", "");
Give G a single control InOut port with annotation ("PR Name", "p0");
Give G a single control OutIn port with annotation ("PR Name", "r0");

// Translate each state into a Call node.
For each state S in CC:
  Create a new Call node C in G;
  If (S has a name)
    add an annotation to C: ("Name", name of S);
  Give C a control input port named "from_prior_state";
  Give C a control output port named "to_next_state";
  Give C a control output port named "to_input_statement";
  Insert the following information as C's second and third parts:
    R.p0 :=
         := R.r0
  If (the name N of the routine to call when entering C is null)
    insert the following into C as its first part:
      R := the RID of NoOpRoutine
  Else if (there is no connected component CC' in the FSA named N)
    error("Undefined action routine");
  Else
    Insert the following into C as its first part:
      // This assumes that the SPG subgraph for CC' has already been
      // generated, and in general this may not be true.  There are ways
      // to solve this problem, e.g., generate only the grouping for CC'
      // now, save the RID for it, then go back later and fill in the
      // rest of the subgraph for CC'.
      R := the RID of the SPG grouping representing CC'

  // Final states become subprogram returns.
  If (S is a final state)
    Give C an annotation ("Return", "");

// Translate each arc into a small subgraph.
For each state S in CC:
  If (there is at least one arc in the FSA originating at S)

    // Check to see if S has an arc leading back to S.
    If (at least one of the arcs leaving S also terminates at S)
      Create a new Loop grouping LG;
      Add to LG the Call node representing S;
      CG := LG;  // CG = current grouping
    else
      CG := G;

    // Set up the basic structure to read the next input value a
    // go to a new state based on that value.
    Create a new Multiassignment node N in CG with one control input
      port named "input" and one data output port named "value"
    Give N the following content
```

```
      value := INPUT
    Create a new NP1 edge in CG and attach its single source branch to
      port "to_input_statement" of the Call node representing S;
      attach its single destination branch (with no condition) to N's
      input port;
    Create a new NPa edge VAL in CG and attach its single source branch
      to port "value" of node N;

    // Now create the edge corresponding to the arcs leading out of S
    Create a new NP1 edge E in CG and attach its single source branch to
      port "to_next_state" of the Call node representing S;
    For each arc A originating at S:
      Create a new destination branch B of E leading to the port
        "from_next_state" on the node corresponding to the state led to
         by A;
      Give B a single data input port named "from_input";
      Create a new conditionless destination branch of edge VAL and
        attach it to port "from_input" of branch B;

      // Create a branch condition corresponding to the inputs enabling
      // travel along A.  Each single value or range of values enabling a
      // transition is called a "component" of the enabling condition.
      For each single-valued component SVC of A's enabling condition add
        the following disjunct to B's condition:
          (from_input = SVC)
      For each range-valued component RVC of A's enabling condition add
        the following disjunct to B's condition:
          ((from_input >= lower value of RVC) and
           (from_input <= upper value of RVC))

      // Add loop annotations if A leads from S to S
      If (A terminates at S)
        Add to B the annotation ("Loop Return", "");
        Create a new unexecutable edge from LG to B with the
          annotation ("Loop Return", "");

    // Add a default branch to a call to the error routine in case the
    // input is invalid
    Create a new Stop Call node CN in CG with a single control input port
      named "from_state" and with the following contents:
        R := the RID of the Error routine for this FSA
        R.p0 :=
             := R.r0
    Create a new destination branch B of E with condition "default" and
      attach it to CN's input port;

// Locate the state in which to begin execution if the subprogram
// representing CC is invoked
If (CC contains a start state SS)
  Add to G a control NP1 edge from p0 of CC to from_previous_state of
    the node corresponding to SS -- put no condition on the edge's
    destination branch;
```

```
Else
  If (there is exactly one node n such that all other nodes in CC are
      reachable from n)
    Add to G a control NP1 edge from p0 of CC to p0 of n -- put no
      condition on the edge's destination branch;
  Else
    error("Can't translate component -- first state to enter is ambiguous");

// The translation of CC is complete -- add it to the SPG
Add G and everything in/on G to the SPG;

// Create a Start node calling G if CC contains a start state.  Control
// will return to the Start node only if CC executes a final state, in
// which case overall execution of the SPG should stop.  Hence the Call
// node is also a Stop node.
If (CC contains a start state SS)
  Create a new Call node CN with the following contents:
    R    := G's RID
    R.p0 :=
         := R.r0
  Give CN a single control input port;
  Give CN an annotation ("Start", "");
  Give CN an annotation ("Stop", "");
  Add CN to the SPG;
```

B.2 Translating Pascal into an SPG

The program that follows doesn't actually add anything to the SPG until the entire Pascal program has been processed. For the batch-translation environment in which this mapping was developed (see Section 7.1), this is not a problem, but for more interactive MVDEs, it would be important to avoid such large-grain modifications to the SPG. For a further discussion of this issue, see Section 8.2.

In places where Pascal expressions are to be written out as SPG expressions, it is implicit that the necessary syntactic transformations take place. Thus, all variable names are replaced with the corresponding VIDs, parentheses in the Pascal are translated into user-specified parentheses in the SPG ($(...)$), and literal values in the Pascal are preceded by a pound sign ("#"). For details on the syntax of SPG expressions, see Section 4.3.1.

```
// Create file grouping and a global scope
Create a new unexecutable Scope grouping GS;
Give GS an annotation ("File", name of source file);
Give GS an annotation ("Name", name of program);

// Often we'll have a current destination branch that's waiting to be
// attached to a node.  CurrentBranch is that branch.
CurrentBranch := null;

// translate global variable declarations
translateVarDecls(GS, CurrentBranch);  // see below

// Translate procedure declarations
For each procedure PROC in the program considered in order:
  Create a new Scope Subprogram grouping PG in GS;
  Give PG an annotation ("Name", name of PROC);

  // create control inputs and outputs for this procedure
  Create a new control InOut port, give it an annotation ("PR Name", "p0"),
    and add it to PG;
  Create a new NP1 edge in PG with a single source branch attached to p0;
  LocalCurrentBranch := a new conditionless destination branch of this new
    edge;
  Create a control OutIn port, give it an annotation ("PR Name", "r0"),
    and add it to PG;

  // translate formal parameters into ports, var decls, and inits
  If (PROC has at least one declared parameter)
    Create a new unexecutable grouping INIT in PG;
    Give INIT an annotation ("Parameter Init", -);
    Create a new Declaration node PARAMS and add it to INIT;
    Give PARAMS a single control input port and a single control output port;
    Attach LocalCurrentBranch to the input port of PARAMS;

    Create a new Multiassignment node INITS and add it to INIT;
    Give INITS a single control input port and a single control output port;
    Create a new NP1 edge and attach its single source branch to the output
```

```
      port on PARAMS and its single destination branch (w/no condition) to
      the input port on INITS;
    Create a new NP1 edge in PG with a single source branch attached to the
      output port of INITS;
    LocalCurrentBranch := a new destination branch of this new edge;

    paramNumber := 1;
    For each parameter PARAM in PROC's parameter list considered in order:
      Create a data InOut port IOP and add it to PG;
      Give IOP an annotation ("PR Name", "p"paramNumber);
      Give IOP an annotation ("Name", name of PARAM);
      Add PARAM's name to the list of variables declared inside PARAMS --
        call the VID for this variable PVID;
      Give INITS a new data input port DIP with
        annotation ("Name", name of PARAM);
      Create a new NP1 edge in PG and attach its single source branch to IOP
        and its single destination branch (w/no condition) to DIP;
      Add to the contents of INITS the following assignment:
        PVID := name of PARAM
      paramNumber++;

  // translate local variable declarations
  translateVarDecls(PG, LocalCurrentBranch);        // see below

  // translate body of procedure
  translateBlock(PG, LocalCurrentBranch);           // see below

  // put a return node at the end of the procedure
  Create a new Epsilon node END in PG with a single control input port;
  Attach LocalCurrentBranch to END's port;
  Give END the Annotation ("Return", "");


// Translate main block
translateBlock(GS, CurrentBranch);                  // see below

// Put a Stop node at the end of the program
Create a new Epsilon node END in GS with a single control input port;
Attach CurrentBranch to END's port;
Give END the Annotation ("Stop", "");

// Now add the whole thing to the SPG
Add GS to the SPG;


// The following routine generates the SPG subgraph corresponding to a VAR
// declaration block.  The parameters of this routine are:
//   CG: The "current" grouping (to which the generated SPG should be added).
//  CDB: The "current" NP1 destination branch.  The code generated by this
//       routine will be led to by CDB.  If CDB is null, no SPG has been
//       generated "above" the subgraph that this routine will generate.
//       This can only happen if we are processing the global variable
```

```
//       declarations, which means we have to make the first node we
//       generate a start node.  The value of CDB is updated by this
//       routine.
procedure translateVarDecls(Grouping CG, DestinationBranch CDB)
  For each "Var" section VS considered in order:
    Create a new Declaration node DN in CG with one input control port and
      one output control port;
    For each variable name V in VS, add V to DN;
    If (CDB == null)
      Give DN an annotation ("Start", "");
    Else
      Attach CDB to the input port of DN:
    Create a new NP1 edge in CG;
    Give the new edge a source branch attached to the output port of DN;
    Give the new edge a destination branch and make CDB that branch;


// The following routine generates the SPG corresponding to a Begin...End
// block.  The parameters of this routine are:
//   CG: The "current" grouping (to which the generated SPG should be added).
//  CDB: The "current" destination branch.  The code generated by this
//       routine will be led to by CDB.  If CDB is null, no SPG has been
//       generated "above" the subgraph that this routine will generate.
//       This can only happen if we are processing the main block, which
//       means we have to make the first node we generate a start node.
//       The value of CDB is updated by this routine.
// Note that a block is never empty -- it always contains at least one
// statement, although that statement may be the empty statement.
procedure translateBlock(Grouping CG, DestinationBranch CDB)
  For each statement S in the block considered in order:
    translateStatement(S, CG, CDB);


procedure translateStatement(Statement S, Grouping CG, DestinationBranch CDB)
  Case (type of S) of:
    compound-statement:
      Create a new unexecutable grouping UG in CG;   // for the block
      translateBlock(UG, CDB);
    conditional-statement:      translateConditional(CG, CDB);
    case-statement:             translateCase(CG, CDB);
    repeat-until-statement:     translateRepeat(CG, CDB);
    for-statement:              translateFor(CG, CDB);
    procedure-call-statement:   translateCall(CG, CDB);
    assignment-statement:       translateAssignment(CG, CDB);
    read-statement:             translateRead(CG, CDB);
    write-statement:            translateWrite(CG, CDB, false);
    writeln-statement:          translateWrite(CG, CDB, true);
    empty-statement:            translateEmpty(CG, CDB);
  end;


// This routine does a straightforward translation.  It does not examine
```

```
// the body of the loop and the termination condition to see if it can
// identify a loop initialization or a loop update.
procedure translateRepeat(Grouping CG, DestinationBranch CDB)
  Create a new Loop Grouping LG in CG;
  EntranceBranch := CDB;
  For each statement S inside the loop considered in order:
    translateStatement(S, LG, CDB);
  Give CDB a condition corresponding to the loop's UNTIL expression;
  Create a new destination branch RETURN with the condition "default";
  E := the edge containing CDB;
  Add RETURN to E;
  Attach RETURN to the same port as EntranceBranch;

  // add loop annotations to loop components
  Add to EntranceBranch:  ("Loop Entrance", "");
  Add to CDB: ("Loop Exit", "");
  Add to E: ("Loop Test", "");
  Add to RETURN:  ("Loop Return", "");

  // add loop annotations to Loop grouping
  Create an unexecutable edge from LG to EntranceBranch and give it the
    annotation ("Loop Entrance", "");
  Create an unexecutable edge from LG to CDB and give it the
    annotation ("Loop Exit", "");
  Create an unexecutable edge from LG to E and give it the
    annotation  ("Loop Test", "");
  Create an unexecutable edge from LG to RETURN and give it the
    annotation ("Loop Return", "");


// This routine translates an Pascal statement of this form:
//   for VARIABLE := EXP1 [down]to EXP2 do STMT
procedure translateFor(Grouping CG, DestinationBranch CDB)
  Create a new Loop Grouping LG in CG;
  EntranceBranch := CDB;

  // generate the subgraph for the loop initialization
  Create a new Multiassignment node INIT in LG with a single control input
    port and a single control output port and with the following content:
      VARIABLE := EXP1
  Attach CDB to INIT's input port;
  Create a new NP1 edge in LG;
  Give the new edge a source branch attached to the output port of INIT;
  Give the new edge a destination branch and make CDB that branch;

  // generate the subgraph for the loop's body
  translateStatement(STMT, LG, CDB);

  // generate the subgraph for the loop update and test
  Create a new Multiassignment node UPDATE in LG with a single control input
    port and a single control output port;
  Attach CDB to INIT's input port;
```

```
  Create a new NP1 edge E in LG;
  Give E a source branch attached to the output port of INIT;
  Give E two destination branches, and make CDB one of those
    branches and RETURN the other one;

  if (this is a "to" loop (as opposed to a "downto" loop))
    Give UPDATE this as its content:
      VID of VARIABLE := VID of VARIABLE + #1
    Give RETURN the following condition:
        VARIABLE <= EXP2
  else // this is a "downto" loop
    Give UPDATE this as its content:
      VID of VARIABLE := VID of VARIABLE - #1
    Give RETURN the following condition:
        VARIABLE >= EXP2

  Attach RETURN to the same port as EntranceBranch;
  Give CDB the condition "default";

  // add loop annotations to loop components
  Add to EntranceBranch:  ("Loop Entrance", "");
  Add to CDB:  ("Loop Exit", "");
  Add to INIT:  ("Loop Init", "");
  Add to UPDATE: ("Loop Update", "");
  Add to E:  ("Loop Test", "");
  Add to RETURN:  ("Loop Return", "");

  // add loop annotations to Loop grouping
  Create an unexecutable edge from LG to EntranceBranch and give it the
    annotation ("Loop Entrance", "");
  Create an unexecutable edge from LG to CDB and give it the
    annotation ("Loop Exit", "");
  Create an unexecutable edge from LG to INIT and give it the
    annotation ("Loop Init", "");
  Create an unexecutable edge from LG to UPDATE and give it the
    annotation ("Loop Update", "");
  Create an unexecutable edge from LG to E and give it the
    annotation ("Loop Test", "");
  Create an unexecutable edge from LG to RETURN and give it the
    annotation ("Loop Return", "");


procedure translateCase(Grouping CG, DestinationBranch CDB)
  // check to see if a prior conditional/case might have preceded us
  If (CDB is conditionless (i.e., has a null condition))
    Edge := the edge to which CDB belongs;
    Remove CDB from Edge;
  else
    // a prior conditional may have led here -- start anew at an Epsilon node
    Create a new Epsilon node EN in CG with one control input port and one
      control output port;
    Attach CDB to EN's input port;
```

```
    Edge := a new NP1 edge in CG with a source branch connected to EN's
      output port;

  // create a set to hold all the destination branches eminating from the
  // statements led to by the case statement
  DestBranches := the empty set;

  // translate each arm of the case statement
  For each arm A of the case statement:
    CDB := a new conditionless destination branch in Edge;
    If (A's case-constant == "otherwise")
      Give CDB the condition "default";
    else
      Give CDB this condition:
          The Case statement's case-index == A's case-constant
    For the statement S led to by A, translateStatement(S, CG, CDB);
    Add CDB to DestBranches;

  // If possible, make all the elements of DestBranches source branches of
  // the same edge
  if (for each branch B in DestBranches, B has no condition and is the only
      destination branch of an edge with exactly one source branch)
    Ports := the empty set;
    For each branch B in DestBranches:
      Add to Ports the port P at which B's edge originates;
      Remove from the SPG the edge attached to P (including its branches);
    Create a new NP1 edge E in CG;
    For each port P in Ports:
      Add to E a source branch that attaches at P;
    Give E a destination branch and make CDB that branch;
  else
    // the branches in DestBranches can't be unified -- insert an epsilon node
    Create a new Epsilon node N in CG and give it one control input port and
      one control output port;
    For each branch B in DestBranches, attach B to N's input port;
    Create a new NP1 edge in CG;
    Give the new edge a source branch attached to the output port of N;
    Give the new edge a destination branch and make CDB that branch;


// This routine translates an Pascal statement of this form:
//   If EXPRESSION Then STATEMENT1 [ Else STATEMENT2 ]
procedure translateConditional(Grouping CG, DestinationBranch CDB)
  // check to see if a prior conditional/case might have preceded us
  If (CDB is conditionless (i.e., has a null condition))
    Edge := the edge to which CDB belongs;
  else
    // a prior conditional may have led here -- start anew at an Epsilon node
    Create a new Epsilon node EN in CG with one control input port and one
      control output port;
    Attach CDB to EN's input port;
    Edge := a new NP1 edge in CG with a source branch connected to EN's
```

```
      output port;
    CDB := a new conditionless destination branch in Edge;

  // translate the if part
  Give CDB a branch condition corresponding to the EXPRESSION part of the
    conditional being translated;
  For the STATEMENT1 part S1 of this conditional,
    translateStatement(S1, CG, CDB);

  // translate the else part (it always exists, though it may be an empty
  // statement)
  Create a new destination branch NDB of Edge with the condition "default";
  For the STATEMENT2 part S2 of this conditional
    translateStatement(S2, CG, NDB);

  // CDB needs to be part of an edge that can be replaced, so if it isn't
  // already dispensible, make it terminate at an Epsilon node and then
  // create a dispensible edge that starts at the new node.
  If (CDB has a non-null condition on it) or
     (the edge containing CDB has more than one destination branch) or
     (the edge containing CDB has more than one source branch)
    insertEpsilon(CG, CDB);

  // Ditto for NDB
  If (NDB has a non-null condition on it) or
     (the edge containing NDB has more than one destination branch) or
     (the edge containing NDB has more than one source branch)
    insertEpsilon(CG, NDB);

  // Now make CDB and NDB source branches of the same edge
  P1 := the port at which CDB's edge originates;
  P2 := the port at which NDB's edge originates;
  Remove from the SPG the edge attached to P1 (including its branches);
  Remove from the SPG the edge attached to P2 (including its branches);
  Create a new NP1 edge in CG with two source branches, one each
    attached to P1 and P2;
  Give the new edge a destination branch and make CDB that branch;


// This routine makes DB terminate at a new Epsilon node and updates DB so
// it refers to the sole destination branch of the edge leaving the new node.
procedure insertEpsilon(Grouping CG, DestinationBranch  DB)
    Create a new Epsilon node N in CG and give it one control input port and
      one control output port;
    Attach DB to N's input port;
    Create a new NP1 edge E in CG;
    Give E a source branch attached to the output port of N;
    Give E a destination branch DB2;
    DB := DB2;


procedure translateCall(Grouping CG, DestinationBranch CDB)
```

```
  Create a new Call node CN in CG with one control input port and one
    control output port and give it the following contents for
    its first and third parts:
      R := the RID of the grouping corresponding to the procedure being called
        := R.r0
  Attach CDB to CN's input port;

  if (there are no actual params)
    make this the content of the 2nd part of CN:
      R.p0 :=
  else
    For each actual param PARAM in position P, considered in order:
      Case (type of PARAM):
        Literal value V:  Add this to the 2nd part of CN:  R.pP := #V
        Variable name V:  Add this to the 2nd part of CN:  R.pP := V's VID
        Expression E:     Add this to the 2nd part of CN:
                            R.pP := E

  Create a new NP1 edge in CG;
  Give the new edge a source branch attached to the output port of CN;
  Give the new edge a destination branch and make CDB that branch;


procedure translateAssignment(Grouping CG, DestinationBranch CDB)
  Create a new Multiassignment node N in CG with one control input port
    and one control output port and with the following content:
      target variable := the right-hand side expression
  Attach CDB to N's input port;
  Create a new NP1 edge in CG;
  Give the new edge a source branch attached to the output port of N;
  Give the new edge a destination branch and make CDB that branch;


procedure translateRead(Grouping CG, DestinationBranch CDB)
  // put everything in a statement grouping if there is more than one
  // variable to read
  If (there is more than one variable to read)
    Create a new Statement grouping SG in CG;
    LCG := SG;   // LCG = Local current grouping
  else
    LCG := CG;

  // now build the subgraph for the reads
  For each variable V to read, considered in order:
    Create a new Multiassignment node N in LCG with one control input port
      and one control output port and with the following content:
        V's VID := INPUT
    Attach CDB to N's input port;
    Create a new NP1 edge in CG;
    Give the new edge a source branch attached to the output port of N;
    Give the new edge a destination branch and make CDB that branch;
```

```
procedure translateWrite(Grouping CG, DestinationBranch CDB, Boolean AddNL)
  // put everything in a statement grouping if there is more than one
  // thing (variable, integer literal, string literal, expression) to write
  If (there is more than one thing to write) or (AddNL == true)
    Create a new Statement grouping SG in CG;
    LCG := SG;   // LCG = Local current grouping
  else
    LCG := CG;

  // now build the subgraph for the writes
  For each thing T to write, considered in order:
    Create a new Multiassignment node N in LCG with one control input port
      and one control output port;
    Case (type of T) of:
      Variable name V:  Give N this content: OUTPUT := V's VID
      Literal string S: Give N this content:
                          OUTPUT := S
      Literal value V:  Give N this content: OUTPUT := #V
      Expression E:     Give N this content:
                          OUTPUT := E
      end;
    Attach CDB to N's input port;
    Create a new NP1 edge in CG;
    Give the new edge a source branch attached to the output port of N;
    Give the new edge a destination branch and make CDB that branch;

  // check to see if a newline has to be added at the end
  If (AddNL == true)
    Create a new Multiassignment node N in LCG with one control input port
      and one control output port and this content:
        OUTPUT := "\n"
    Attach CDB to N's input port;
    Create a new NP1 edge in CG;
    Give the new edge a source branch attached to the output port of N;
    Give the new edge a destination branch and make CDB that branch;


procedure translateEmpty(Grouping CG, DestinationBranch CDB)
  Create a new Epsilon node N in CG and give it one control input port and
    one control output port;
  Attach CDB to N's input port;
  Create a new NP1 edge in CG;
  Give the new edge a source branch attached to the output port of N;
  Give the new edge a destination branch and make CDB that branch;
```

B.3 Translating an SPG into an FSA

I explained in Section 7.4 that the input to this algorithm is an SPG subgraph corresponding to a subprogram or an STask, etc. In the pseudocode that follows, I refer to this subgraph as TSPGS, "The SPG Subgraph."

I assume that there is an auxiliary data structure with the following fields for each FSA arc:

- **SPGstart**: The SPG port where the arc originates.
- **SPGend**: The SPG port where the arc terminates.
- **FSAstart**: The FSA state where the arc originates.
- **FSAend**: The FSA state where the arc terminates.
- **values**: The set of value ranges enabling traversal of the arc.
- **subgraph**: The set of components in TSPGS corresponding to this arc.

Similarly, I assume that there is an auxiliary data structure with the following fields for each FSA state:

- **inArcs**: The set of arcs terminating at this state.
- **outArcs**: The set of arcs originating at this state.
- **actionRoutine**: The (TID, RID) pair for the routine to be invoked when this state is entered.
- **initialState**: True iff this state is the FSA's initial state.
- **finalState**: True iff this state is a final state of the FSA.
- **subgraph**: The set of components in TSPGS corresponding to this state.

All references to SPG nodes, edges, and ports in this algorithm are to executable nodes, edges, and ports; unexecutable components are not considered when trying to make sense of TSPGS as an FSA.

```
// first we look for TSPGS subgraphs that look like FSA arcs
arcs := the empty set;
findArcs();

// next we look for TSPGS subgraphs that look like FSA states
states := the empty set;
if (the cardinality of arcs > 0)
  findStates();
else
  // we didn't find any arcs, so make all of TSPGS a single FSA state
  create a new state S;
  S.inArcs := the empty set;
```

```
  S.outArcs := the emtpy set;
  S.subgraph := TSPGS;
  add S to states;

// At this point, the states know which arcs attach to them, but not vice
// versa.  Rectify this inconsistency.
for each arc A in arcs:
  P := the port in A.SPGstart;
  if (there exists a state S such that S.subgraph contains P)
    A.FSAstart := S;
  P := the port in A.SPGend;
  if (there exists a state S such that S.subgraph contains P)
    A.FSAend := S;

// It's time now to look for an FSA start state
for each state S in states:
  S.initialState := false;

if TSPGS is a Subprogram grouping G
  // Look for a node that must be executed first when G is invoked
  if (G has an InOut port IOP with PR Name "p0") and
     (at least one source branch in G is attached to IOP) and
     (all destination branches of all edges containing source branches
      attached to IOP are attached to ports on a single node N) and
     (N is contained in S.subgraph for some state S)
    S.initialState := true;
    if (N is a Call Node)
      setActionRoutine(N, S);
  else
    // look for a node that must be executed immediately after
    // parameters have been initialized
    if (there exists exactly one Parameter Init grouping PIG in G)
      edgesFromPIG := every edge in TSPGS that contains a source branch
                      attached to a port in PIG and contains a destination
                      branch attached to a port outside PIG;
      if (there exists a node N in TSPGS such that every destination branch of
          every edge in edgesFromPIG is attached to a port on N) and
         (N is contained in S.subgraph for some state S)
    S.initialState := true;
    if (N is a Call Node)
      setActionRoutine(N, S);

else  // TSPGS is not a Subprogram grouping
  if there is exactly one Start node SN in TSPGS
    if there exists a state S such that S.subgraph contains SN
      S.initialState := true;
      if (SN is a Call Node)
        setActionRoutine(SN, S);

// Look for FSA accepting states
for each state S in states:
  if S.subgraph contains at least one Return node or at least one Stop node
```

```
    S.finalState := true;
  else
    S.finalState := false;

// now we try to identify the action routines for the states of the FSA
for each state S in states:
  OldAR := S.actionRoutine;  // this may have been set when looking for
                             // a start state

  routineSpecs := the empty set;
  allNodesAgree := true;
  for each arc A in S.inArcs:
    if (A.SPGend is on a Call node CN) and (CN is in S.S.subgraph)
      if (the TID specification part of CN is missing) or
         (the TID specification part of CN can be statically evaluated)
        if (the RID specification part of CN can be statically evaluated)
          routineSpecs += ( the value of CN's TID specification,
                            the value of CN's RID specification );
        else
          allNodesAgree := false;
          break;
      else
        allNodesAgree := false;
        break;
    else
      allNodesAgree := false;
      break;

  if (allNodesAgree) and (|routineSpecs| == 1)
    A := some arc in S.inArcs;
    setActionRoutine(the node containing A.SPGend, S);

  // make sure the current action routine is consistent with the one we
  // came in with
  if (OldAR != UNDEFINED) and (OldAR != S.actionRoutine)
    S.actionRoutine := UNDEFINED;

// Finis!  Display the FSA.  Unconstrained arcs lead to the implicit error
// state, so they are not displayed.
For each state S in states:
  display S;
For each arc A in arcs:
  if A.values is constrained
    display A;


// Identify subgraphs that look like FSA arcs.  Start by looking for the
// right kind of Multiassignment nodes.  There are two kinds to look for,
// one using data flow, the other using control flow.  We look for the data
// flow variety first.
procedure findArcs()
  For each Multiassignment node N in TSPGS:
```

```
// see if N has a dataflow edge that controls token flow on another edge
If (N has a single output port OP) and
   (OP is a data port) and
   (N's content is of the form "OP := INPUT") and
   (there is an NPa edge E1 attached to OP) and
   (E1 has exactly one source branch) and
   (E1 has at least one destination branch) and
   (each of E1's destination branches has no condition)

  // N and E1 have the right topology -- see if E1 controls token flow
  // along another edge
  If (there exists an NP1 or Select edge E2 such that
      E2 has at least two destination branches and
      one of those branches has the condition "default" and
      each of E2's destination branches that does not have the
      condition "default" has a data input port to which a branch of E1
      attaches)

    // E2 has the right topology and relationship to E1.  Now make sure
    // that E2 and edges leading to N come from the same node.
    portSet := the set of ports to which source branches of E2 are
               attached;
    for each edge E with a destination branch terminating at a port on N:
      add to portSet each port to which a source branch of E is attached;
    if (all the ports in portSet belong to the same node)

      // Topologically we are in good shape.  Set up pending arcs for
      // each of E2's destination branches.
      pendingArcs := the empty set;
      for each destination branch DB of E2:
        Create a new arc A;
        A.SPGstart := some randomly chosen port in portSet;
        A.SPGend := port to which DB is attached;
        A.FSAstart := UNDEFINED;
        A.FSAend := UNDEFINED;
        A.subgraph := N + E1 + E2's source branches + DB;
        for each edge E leading to an input port on N:
          add E's source branches to A.subgraph;
          for each destination branch B in E attached to an input port
            on N:
              add B to A.subgraph;

      // All that remains is to verify that the conditions on E2's
      // destination branches "look like" values on FSA arcs and to
      // assign a value range to the arc represented by each branch.

      // The calls to "List" below simply indicate that the given data
      // is being put in the proper order for a the call to verifyArcsDF.
      // See the description of verifyArcsDF for details on this order.
      if (verifyArcsDF(List(E2's destination branches),
                       List(ports on E2 to which E1's branches attach),
                       List(pendingArcs)))
```

```
        add the contents of pendingArcs to arcs;

  // The Multiassignment node N didn't look like an arc based on
  // dataflow, but we now have to see if it looks like an arc based on
  // control flow.
  else
    if (N has a single output port OP) and
       (OP is a control port) and
       (N has a single input port) and
       (N's content is of the form "VID := INPUT" for VID VID) and
       (there is an NP1 or Select edge E attached to OP) and
       (E has at least two destination branches) and
       (all of E's destination branches have a non-null condition) and
       (one of E's destination branches has the condition "default")

      // N has the right topology and content and E has the right
      // topology.  Set up pending arcs for each of E2's destination
      // branches.
      pendingArcs := the empty set;
      for each destination branch DB of E:
        Create a new arc A;
        A.SPGstart := N's input port;
        A.SPGend := port to which DB is attached;
        A.FSAstart := UNDEFINED;
        A.FSAend := UNDEFINED;
        A.subgraph := N + E - (N's input ports);

      // All that remains is to verify that the conditions on E's
      // destination branches "look like" values on FSA arcs and to
      // assign a value range to the arc represented by each branch.
      if (verifyArcsCF(List(E's destination branches),
                       VID,
                       List(pendingArcs),
                       [-infinity, +infinity]))

        // Everything in pendingArcs is in fact an arc now, but we have
        // to check each arc with an unconstrained set of values to see
        // if the SPG subgraph representing it can be extended by passing
        // through Epsilon nodes.  If so, we may be able to discover more
        // arcs, possibly with constrained values.  This kind of SPG
        // topology arises from cascaded if-then-else constructs, as
        // discussed above.  The cascading may be arbitrarily deep, so
        // we have to keep doing this check until we can't extend an arc
        // any longer.
        repeat
          somethingChanged := false;
          for each arc A1 in pendingArcs such that A1.values is
          unconstrained:
            if (the port in A1.SPGend is the only input port on an Epsilon
                node EN) and
               (EN has only one output port OP) and
               (there is an NP1 or Select edge E attached to OP) and
```

```
                (E has at least two destination branches) and
                (all of E's destination branches have a non-null
                 condition) and
                (one of E's destination branches has the condition "default")

              // again, we have the right topology, so we have to check the
              // branch conditions
              morePendingArcs := the empty set;
              for each destination branch DB of E:
                create a new arc A2 that is a copy of A1;
                A2.SPGend := port to which DB is attached;
                A2.subgraph += EN + DB + E's source branch;

              // All that remains is to verify that the conditions on E's
              // destination branches "look like" values on FSA arcs and to
              // assign a value range to the arc represented by each branch.
              if (verifyArcsCF(List(E's destination branches),
                               VID,
                               List(morePendingArcs),
                               A1.values))

                // if all the value ranges of the arcs in morePendingArcs
                // are subsets of A1.values, we can replace A1 with the
                // contents of morePendingArcs
                legalValues := true;
                for each arc A3 in morePendingArcs:
                  if (there exists a value V such that A3.values accepts
                      V and A1.values does not accept V)
                    legalValues := false;

                if (legalValues == true)
                  remove A1 from pendingArcs;
                  pendingArcs += morePendingArcs;
                  somethingChanged := true;

        until (somethingChanged == false);

        // finally the pending arcs can become full-fledged arcs
        add the contents of pendingArcs to arcs;


// This function takes a list of destination branches, a list of data input
// ports, and a list of arcs such that the same conceptual object is in the
// same position in the three lists.  That is, the destination branch
// DBL[i] has a data input port DIPL[i] and is part of the subgraph
// representing the arc AL[i].  If a branch in DBL has the condition
// "default", the corresponding position in PL is null.
//
// The function checks each branch condition to see if it "looks like" the
// kind of condition we expect to see on an FSA arc, and if so, it fills in
// the value field of each arc in AL and returns true.  Otherwise it
// returns false.
```

```
//
// Because we are implementing deterministic FSAs, this function also
// returns false if there is an overlap in the allowed values on the
// destination branches.
function verifyArcsDF(List_of_Destination_Branch DBL,
                      List_of_Data_Input_Ports DIPL,
                      List_of_Arc AL) return boolean
  for i := 1 to number of list elements do:
    // reject conditions that are null, don't involve the branch's input
    // port, or can't be statically evaluated
    if the condition C controlling DBL[i] is null
      return false;
    if C is "default"
      AL[i].values = {[-infinity, +infinity]};
      defaultBranch := i;
    else
      for each conjunct and disjunct in C (i.e., each LogicalExp in the grammar
      of Figure 4-23 of the thesis):
        if a name for DIPL[i] does not comprise the entire expression on one
        side of the relational operator
          return false;
        if the expression being compared with the value on DIPL[i] cannot be
        statically evaluated by looking only at C
          return false;

      // we can now do the following because we know that all the constraints
      // on the value of DIPL[i] can be statically evaluated
      examine C and determine
        VR = the ranges of values such that if the value of the token on
             DIPL[i] is within VR, C will be true;
      AL[i].values := VR;

  // check to make sure there is no overlap in the values of the arcs
  for i := 1 to number of list elements do:
    for j := 1 to number of list elements do:
      if (i != defaultBranch) and (j != defaultBranch) and (i != j)
        if (there is any overlap in AL[i] and AL[j])
          return false;

  // The default branch is taken only if all other branch conditions
  // evaluate to false, so remove from the value range for the default
  // branch the valid ranges of all the other branches.  The result may be
  // either a constrained or an unconstrained value range.
  for i := 1 to number of list elements do:
    if i != defaultBranch
      A[defaultBranch].values -= A[i].values;


// This function takes a list of destination branches, a VID, and a list of
// arcs such that the same conceptual object is in the same position in the
// two lists.  That is, the destination branch DBL[i] is part of the
// subgraph representing the arc AL[i].  VID is the VID of the variable
```

```
// whose value is expected to control token flow along the branches in DBL.
//
// The function checks each branch condition to see if it "looks like" the
// kind of condition we expect to see on an FSA arc, and if so, it fills in
// the value field of each arc in AL and returns true.  Otherwise it
// returns false.
//
// Because we are implementing deterministic FSAs, this function also
// returns false if there is an overlap in the allowed values on the
// destination branches.
function verifyArcsCF(List_of_Destination_Branch DBL,
                      Variable_Identifier VID,
                      List_of_Arc AL,
                      Value_Range TotalVR) return boolean
  for i := 1 to number of list elements do:
    // reject conditions that are null, don't involve the branch's input
    // port, or can't be statically evaluated
    if the condition C controlling DBL[i] is null
      return false;
    if C is "default"
      AL[i].values = TotalVR;
      defaultBranch := i;
    else
      for each conjunct and disjunct in C (i.e., each LogicalExp in the grammar
      of Figure 4-23 of the thesis):
        if VID does not comprise the entire expression on one
        side of the relational operator
          return false;
        if the expression being compared with VID cannot be
        statically evaluated by looking only at C
          return false;

      // we can now do the following because we know that all the constraints
      // on the value of the variable represented by VID can be statically
      // evaluated
      examine C and determine
        VR = the ranges of values such that if the value of the variable
             represented by VID is within VR, C will be true;
      AL[i].values := VR;

  // check to make sure there is no overlap in the values of the arcs
  for i := 1 to number of list elements do:
    for j := 1 to number of list elements do:
      if (i != defaultBranch) and (j != defaultBranch) and (i != j)
        if (there is any overlap in AL[i] and AL[j])
          return false;

  // The default branch is taken only if all other branch conditions
  // evaluate to false, so remove from the value range for the default
  // branch the valid ranges of all the other branches.  The result may be
  // either a constrained or an unconstrained value range.
  for i := 1 to number of list elements do:
```

```
    if i != defaultBranch
      A[defaultBranch].values -= A[i].values;


// Identify subgraphs that look like FSA states.  The general strategy is
// straightforward: given n arcs, simultaneously perform 2n breadth-first
// graph traversals, one starting at each arc endpoint, and continue each
// traversal until it has nowhere to go that no other traversal has gone
// before.  The extent of each traversal then corresponds to the subgraph
// of an FSA state.  When two traversals bump into one another, attempt to
// merge the putative states they are discovering.
//
// There are two situation where merging states is not allowed.  The first
// is when two traversals belong to the same FSA arc (i.e., correspond to
// searches initiated at opposite ends of the same arc).  This is because
// the kinds of subgraphs that identify FSA arcs are expected to be
// relatively rare, and it would not be uncommon for an SPG to be fully
// connected even when the subgraphs corresponding to such arcs were
// removed from the SPG.  If that were the case, only a single FSA state S
// would be discovered, and all arcs would go from S to S.  To prevent
// this, an arc A is allowed to both originate and terminate at the same
// state S only if A.subgraph and the connection points (ports) between A
// and S are all contained inside the same Loop grouping.  If this
// constraint would be violated by the merging of two putative states, the
// states are not merged.
//
// The second situation in which we don't merge states is when the result
// of that merge would be a state where two or more arcs could be enabled
// by the same input value.
procedure findStates()
For each arc A in arcs:
  // we don't search the parts of TSPGS led to by unconstrained arcs,
  // because they presumably lead to the implicit error state
  if A.values is constrained

    // create a new putative state for each arc endpoint
    create a new state S1;   // for the state where A originates
    S1.inArcs := null;
    S1.outArcs := { A };
    S1.actionRoutine := UNDEFINED;
    S1.subgraph := { A.SPGstart };
    add S1 to states;

    create a new state S2;   // for the state where A terminates
    S2.inArcs := { A };
    S2.outArcs := null;
    S2.actionRoutine := UNDEFINED;
    S2.subgraph := { A.SPGend };
    add S2 to states;

// It is possible for a single input port to be at the end of more than one
// arc, but we only need a single state there, because there are no
```

```
// restrictions in arcs entering a state.  The following loop thus merges all
// putative states consisting of the same input port.
for all states S1 in states:
  P := the element in S1.subgraph;
  for all states S2 in states such that S2 != S1;
    if (the element in S2.subgraph == P)
      mergeStates(S1, S2);

// set up a BFS for each putative state
for each state S in states:
  create an empty queue of ports (portQueue) for S;
  P1 := the port in S.subgraph;
  add P1 to S.portQueue;

  // S.portQueue will hold ports such that the branch(es) connected to each
  // port should be traversed during a search step.  The only port we have
  // now, however, is the one in S.subgraph, which is a port at the *end*
  // of a branch (because the branch belongs to an arc).  To prevent the
  // state search from failing at the outset, we have to see if we can add
  // to S.portQueue the other ports on the SPG component containing the
  // port in S.subgraph.
  C := the SPG component of which P1 is a part;
  if (C is a node) and
     (C is not in a subgraph of an element of arcs)
    add C to S.subgraph;
    for each port P2 on C such that P2 != P1:
      if (P2 is in S2.subgraph for some state S2) and (S != S2)
        if (okayToMerge(S, S2)
          mergeStates(S, S2);
        else
          remove P2 from S.subgraph;
      else
        add P2 to S.portQueue;

// now cycle through the states, performing a single BFS traversal step each
// time around the loop
repeat
  didSomething := false;
  for each state S1 in states:
    if S1.portQueue is not empty
      remove the port P1 at the front of S1.portQueue;
      didSomething := true;
      for each edge E containing a branch connected to P1:
        if (there exists an arc A such that a branch of E is in A.subgraph)
          // do nothing -- we've bumped up against
          // a (possibly unconstrained) arc
        else if (there exists a state S2 such that E is in S2.subgraph) and
                (S1 != S2)
          if (okayToMerge(S1, S2) == true)
            mergeStates(S1, S2);

        else
```

```
          // E isn't part of the FSA yet -- add it to S1
          add E to S1.subgraph;
          for each port P to which a branch of E is attached:
            if (P is not in a subgraph of an element of arcs or states)
              add P to S1.subgraph;

              C := the SPG component of which P is a part;
              if (C is a node) and
                 (C is not in a subgraph of an element of arcs or states)
                add C to S1.subgraph;
                for each port P2 on C:
                  if (P2 is not in a subgraph of an element of arcs or states)
                    add P2 to S1.portQueue;

until (didSomething == false);


function okayToMerge(State S1, State S2) return boolean
  // it doesn't make a lot of sense, but it's always okay to merge a state
  // with itself
  if (S1 == S2)
    return true;

  sameArcs := the intersection of S1.inArcs and S2.outArcs;
  sameArcs += the intersection of S2.inArcs and S1.outArcs;

  if (sameArcs is the empty set)
    // no arc runs between S1 and S2; check to see if the outgoing
    // arcs have disjoint enabling conditions
    for each arc A1 of S1.outArcs such that A1.values is constrained:
      for each arc A2 of S2.outArcs such that A2.values is constrained:
        if (there exists a value V such that V is in both
            A1.values and A2.values)
          return false;

    return true;

  else
    // sameArcs has at least one arc in it that runs between S1
    // and S2;  merging is okay only if each arc and its connecting
    // points is inside a Loop grouping
    for each arc A in sameArcs:
      if (there exists a Loop grouping LG such that
          each component of A.subgraph is contained in LG and
          A.SPGstart is contained in LG and
          A.SPGend is contained in LG)
        // do nothing
      else
        return false;

    return true;
```

```
procedure mergeStates(State S1, State S2)
  // there's no point in merging a state with itself
  if (S1 != S2)
    S1.inArcs += S2.inArcs;
    S1.outArcs += S2.outArcs;
    S1.subGraph += S2.subGraph;
    S1.portQueue += S2.portQueue;
    remove S2 from states;


// if the routine to be invoked from CN can be statically determined, this
// routine sets S.actionRoutine to that routine.  Otherwise it has no effect.
procedure setActionRoutine(Call_Node CN, State S)
  if (there is an STask specification SS in the first part of CN)
    if (the value of SS can be evaluated statically)
      TID := the value of SS;
    else
      return;   // STask to call is undefined
  else
    TID := null;

  if (the RID of the routine to be invoked by CN is an expression whose
      value can be evaluated statically)
    RID := the value of that expression;
  else
    return;   // Routine to call is undefined

  S.actionRoutine := (TID, RID);
```

B.4 Translating an SPG into Pascal

The program below makes calls to a routine called emit. This routine, which is called like the Pascal write routine, adds code to the view, i.e., it is the mechanism through which the translator indicates what the translation of an SPG should be. The routines startComment and endComment are used to demarcate comment boundaries; all emitted text within such a region is written as a comment in the concrete representation of the program.

In the code that follows, a *simple edge* is an NP_1 control edge with exactly one source branch and exactly one destination, and the destination branch is restricted to having no condition on it. A *simple data edge* is a simple edge for data tokens.

```
// We use the following global variable to keep track of the SPG node most
// recently translated into Pascal.  (It facilitates the detection of loop
// entries.) Of course, this variable could be eliminated through better
// parameterization, but, like software in real life, sometimes a kludge is
// simply the most cost-effective short-term way to get things done.
lastNode := null;

// We use the global stack variable "loops" to keep track of loops we're
// inside of.  Each element of loops must be either FOR or REPEAT,
// reflecting the type of loop we're inside of.  There are three operations
// defined for loops:  push, pop, and top, the latter function returns the
// element at the top of the stack without returning it.
loops := the empty stack;

// STasks will be ignored, because they make no sense in Pascal, but we do
// generate a comment for each one listing its entries.
for each STask grouping STG in the SPG:
  startComment();
  emit("There is an STask grouping ");
  emit(nameOf(STG));
  emit(" in the SPG.  Entries are:\n"):
  for each Entry grouping EG in STG:
    emit(nameOf(EG), "\n");
  endComment();

// We start with the pro forma stuff at the beginning of the program.
emit("program ");
groupingSet := set of all groupings G such that
               (all executable components of the SPG are contained
                inside G) and
               (G has an annotation named "Name");
if (groupingSet is the empty set)
  create a new grouping GLOBAL;
  for each executable component EC of the SPG:
    add EC to GLOBAL;
  generate a name N of the form PASCALnnn for GLOBAL;
  give GLOBAL an annotation ("Name", N);
  add GLOBAL to the SPG;
else
```

```
  GLOBAL := the grouping in groupingSet with the lexicographically smallest
            value for the "Name" annotation;
emit(the value of GLOBAL's "Name" annotation);
emit("(input, output);");

// Now we declare global variables.
declareVariables(the SPG subgraph from which STask and Subprogram groupings
                 (and their contents) have been removed);

// Next we declare global procedures.
for each Subprogram grouping SG in the SPG such that SG is not contained
  inside a Subprogram or STask grouping:
    // if it looks like a function, just list its name
    if (SG doesn't have exactly one OutIn port OIP) or
       (OIP is a data port) or
       (OIP lacks the PR name "r0")
      startComment();
      emit("There appears to be a function called ", nameOf(SG),
           " in the program.\n");
      endComment();
      continue;    // move on to next subprogram grouing

    // okay, what we have looks like a procedure (so far)
    emit("procedure ")
    nameOf(SG);
    // now generate a parameter list
    params := set of InOut ports on SG;
    if (params has zero members) or
       (there is a member of params with no PR name) or
       (the PR names of the member of params don't form a sequence
        p0, p1, p2, ... pn for n = |params|-1) or
       (the member of params with PR name p0 isn't a control port) or
       (the other members of params aren't data ports)
      mark SG as having a confusing parameter list;
      declareConfusingParamList();
    else if (there is only one element in params)
       // it must be p0, meaning no declared params, which, thanks to
       // Pascal's lovely syntax, means no parameter list
    else
       // we have a normal parameter list -- generate it
       emit("(");
       for I := 1 to (|params|-1) do:
         P := the element of params with PR name pI;
         nameOf(P);
         if (I < (|params|-1)
           emit(", ");
       emit(")");

    // and finally we can end the parameter list declaration
    emit(";");

    // now we declare local variables
```

```
declareVariables(the contents of SG (excluding SG itself));

// Nested routines will also be ignored, because they don't exist in
// our subset of Pascal, but we do generate a comment for each one
// listing its entries.
listNestedRoutines(SG, true);

// If the parameter list makes no sense, there's no point in trying to
// continue with this routine.  Without a well-formed parameter list,
// it will be impossible to find the initial point of control flow, to
// match actual and formal parameters, and/or to determine how to
// initialize local variables acting as formal parameters.  As they say
// in the AT&T commercials, it's just not worth it.
if (SG has more than one InOut port) and
   (SG is marked as having a confusing parameter list)
  giveUp();
  continue;     // move on to next Subprogram grouping

// locate the port from which control flow in this routine starts
P0 := the InOut control port on SG with the PR name "p0";
if (there is no simple edge attached to P0)
  giveUp();
  continue;      // move on to next Subprogram grouping
else
  initialEdge := the edge attached to P0;

// make sure each of the parameters is initialized in the conventional
// manner
if (|params| > 1)
  if (there is no unique Parameter Init grouping PIG in SG) or
     (there is no Declaration node DN in PIG such that a simple edge
      leads from P0 to DN) or
     (there is no Multiassignment node MN in PIG such that a simple
      edge leads from DN to MN) or
     (there is no node N outside PIG such that a simple edge
      initialEdge leads from MN to N)
    giveUp();
    continue;     // move on to next Subprogram grouping
  else
    // check for simple data edges from non-p0 InOut ports on SG to MN
    somethingsWrong := false;
    for each port P in params:
      if (P is a data port) and
         ((there is no simple data edge from P to a port DP on MN) or
          (more than one edge leads to DP))
        somethingsWrong := true;
    if (somethingsWrong)
      giveUp();
      continue;     // move on to next Subprogram grouping

    // topologically we are in good shape -- check the contents of DN
    // and MN
```

```
        somethingsWrong := false;
        for each data InOut port P:
            if (P has no "Name" annotation) somethingsWrong := true;
            if (there is no variable V in DN with the same name as the
                value of P's "Name" annotation)
              somethingsWrong := true;
            else
              VID := V's VID;
            if (VID is not the target of exactly one assignment in MN) or
               (the expression on the right-hand side of the assignment is
                not the name of a data input port)
              somethingsWrong := true;

        for each variable V in DN:
          if (there doesn't exist an element E of params such that E
              has a "Name" annotation whose value is the same as the name
              of V)
            somethingsWrong := true;

        for each assignment A in MN:
          if (the target of A isn't the VID of a variable declared in DN) or
             (the right-hand side expression of A is other than the name of
              a data input port)
            somethingsWrong := true;

        if (somethingsWrong)
          giveUp();
          continue;     // move on to next Subprogram grouping

    // Everything looks okay so far, and N is the node where control
    // flow starts.  Generate Pascal for this routine.
    generateProcedure(SG, initialEdge);

// We are finally in a position to generate the code for the main block
generateMainBlock(GLOBAL);


// This routine emits the begin...end block at the outermost scope of a
// procedure, and it calls generate to fill in everything in between.
// Unless, that is, the SPG in SG isn't connected, in which case we just
// bail immediately.
procedure generateProcedure(Subprogram_Grouping SG, SimpleEdge E)
  emit("begin");

  // make sure that this subprogram has exactly one return point;  more or
  // fewer is illegal in Pascal
  returns := 0;
  for each node N in SG:
    if (N is not in an STask grouping) and
       (N is not in a Subprogram grouping that is itself inside SG) and
       (N is a Return node)
      returns++;
```

```
  if (returns != 1)
    giveUp(true);

  checkForDeadCode(SG, the node to which E leads);
  generate(E);
  emit("end;");


// This routine emits the begin...end block at the outermost scope of a
// program, and it calls generate to fill in everything in between.  To
// find the initial flow of control, it looks for a unique Start node in
// the grouping G containing the SPG corresponding to the main block in
// Pascal.
procedure generateMainBlock(Grouping G)
  emit("begin");
  startNodes := the empty set;
  for all nodes N in G;
    if (N is a Start node) and
       (N is not in an STask grouping)
      add N to startNodes;

  if (|startNodes| != 1)
    giveUp(true);
  else
    startNode := the sole element of startNodes;

    if (startNode has more than one control input port CIP)
      giveUp();
    else
      if (there is an edge E attached to CIP) and
         (E has more than one source))
        giveUp();

      checkForDeadCode(G, startNode);
      generate(startNode);
  emit("end.");


// Because there is no "return" statement in Pascal, it is essentially
// impossible to have code that is *physically* unreachable (as opposed to
// logically unreachable).  If G contains any physically unreachable code,
// i.e., if the executable part of the SPG in G is unconnected, quit now.
procedure checkForDeadCode(Grouping G, Node initNode)
  for each node N in G such that N != initNode:
    if (N is not in an STask grouping) and
       (N is not in a Subprogram grouping that is itself inside G) and
       (there is no directed path through G from initNode to N)
      giveUp(true);


// This is the primary routine used to translate the SPG into a series of
// Pascal statements. startLocation tells where in the graph to start
```

```
// translating.  If startLocation is an edge, everything upstream from the
// edge has already been translated.  If startLocation is a node, it must
// be a Start node, because there is no edge entering it.  (If there were,
// that edge would have been passed instead.)
//
// If startLocation is an NPn or NPa edge, the parameter startBranch has
// meaning.  All we usually care about when we enter this routine is the
// edge to follow, because we expect to see nothing but NP1 edges in the
// graph.  Such edges have branch conditions that result in the edges being
// translated into if/the/else or case statements if there is more than one
// destination branch.  However, we also want to allow NPa and NPn edges if
// all but one branch leads to Multiassignment nodes performing
// computations to be used to control branch flow (see the routine
// "translateNode" for details).  In that case only, this routine needs to
// know which branch to follow, because we only want to generate code for
// the control path corresponding to that branch.  startBranch identifies
// that branch.  The implementation that follows rejects NPa and NPn edges,
// but the interface to this routine is designed to allow for expansion in
// this area.  If startLocation is an NP1 edge or is a node, the value of
// startBranch is ignored (and is typically null).
//
// The code downstream from a conditional must be generated only once, but
// all branches of the conditional lead to this downstream code.  How then
// to know whether to generate the code or not, given an edge/branch to
// follow?  That's controlled by the parameter quitAtJoinPoint, which is
// true by default, because most of the time we do NOT want to follow a
// branch that represents a join point.  In particular, we only want to
// generate code for the downstream portion of the SPG after generating the
// code for *all* branches of the conditional that precede it.  Thus we
// only set quitAtJoinPoint to false after the "else" branch of an
// if/then/else and after the last arm of a case statement.
//
// This particular implementation of this routine  happens to be recursive.
// An iterative implementation would be be preferable in practice, but it
// would not fundamentally differ from the approach to code generation
// embodied here.
procedure generate(SPG_Component startLocation,
                   DestinationBranch startBranch = null,
                   Boolean quitAtJoinPoint = true)
  // set CE to the current edge to be translated, CDB to the current
  // destination branch to follow
  if (startLocation is a node)
    (CE, CDB) := translateNode(startLocation, null);
  else
    (CE, CDB) := (startLocation, startBranch);

  // if CE is null, there is no edge to follow, so we just return immediately
  if (CE == null)
    return;

  // if this isn't an NP1 edge, just punt right now.  This restriction
  // could be relaxed -- see the comments above.
```

```
if (CE isn't an NP1 edge)
  giveUp(true);

// if CE has more than one source branch, it is a join point (e.g, where
// the code segments for the "if" and "else" clauses of a conditional come
// together).  The code beyond such a point must be generated exactly
// once, so we don't generate it unless we are explicitly allowed to.
if (CE has more than one source branch) and (quitAtJoinPoint)
  return;

// If CE has only a single destination branch, we have simple sequential
// control flow, and we can just translate the node and continue.
// However, we still have to check for some unexpected graph topologies.
if (CE has exactly one destination branch)
  DB := CE's destination branch;
  if (DB has a condition on it)
    giveUp(true);
  else
    // If this is a loop return branch, this must be the end of a for
    // loop, so generate the "end" that matches the "begin" we already
    // emitted, then return.  We don't follow the branch, because we've
    // already translated into Pascal the node where it leads.
    if (DB has an annotation named "Loop Return")
      emit("end");
      return;
    else
      currentNode := the node containing the port to which DB is attached;
      generate(translateNode(currentNode, DB));

// if CE has two destination branches with disjoint and exhaustive
// conditions, we have an if statement or the end of a loop.
else if (CE has two destination branches)
  if (neither of CE's destination branches has the condition
      "default") or
     (one or both of CE's destination branches has no condition)
    giveUp(true);
  else
    translateIf(CE);

// CE has more than two destination branches
else
  // We have either nested if/then/else statements or we have a
  // case statement.  First try to generate a case statement.
  caseOK := true;

  // All conditions must be of one of these forms:
  //   Variable Expression == Constant Expression
  //   Constant Expression == Variable Expression
  //   default
  // In theory, the variable expression might contain references to
  // data ports, but in this mapping I don't allow that.
  VEs := the empty set;
```

```
    for each destination branch DB in CE:
      if (DB has an input port)
        caseOK := false;
        break;

      // make sure that DB's branch condition is of a form compatible with
      // a case statement
      if (DB's condition is not "default")
        if (DB's condition is not a LogicalExp (as defined in the grammar
            in Figure 4-23 of the thesis)) or
           (the operator in the LogicalExp is not "==")
          caseOK := false;
          break;

      // We know we have an expression of this form:
      //   Expression == Expression
      // We have to ensure that it really has one of these forms:
      //   Variable Expression == Constant Expression
      //   Constant Expression == Variable Expression
      if (exactly one of the expressions begin compared for equality can be
          statically evaluated)
        VEs += the expression that cannot be statically evaluated;
      else
        caseOK := false;

    // if VEs is a singleton set, we have a case statement, and the
    // lone element of the set is the index expression.
    if ((caseOK) and (|VEs| == 1))
      // generate a case statement
      translateCase(CE, the sole element of VEs);
    else
      // if possible, generate a series of if/then/else statements
      if (one of CE's destination branches has the condition "default")
        translateIf(CE);
      else
        giveUp(true);


// This function translates the given node into Pascal and then returns the
// SPG edge and destination branch that should be followed to continue
// generating code.  If no code should follow this node (because it is a
// Return or Stop node), the tuple (null,null) is returned.  The parameter
// DB is the destination branch we are following into this node;  if there
// is no such branch (as, for example, at the beginning of a program), it
// is null.  We need it so we avoid generating code for a loop more than
// once.  If we arrive at N via a Loop Return branch, we return immediately.
function translateNode(Node N, DestinationBranch DB)
  return (Edge, DestinationBranch)

  // Return immediately if we got here via a Loop Return node.
  if (DB is not null) and (DB has an annotation named "Loop Return")
    return (null, null);
```

```
// verify the legitimate topology of the current node
checkNode(N);

// Check to see if we just entered a loop.  If so, figure out what kind.
if (lastNode != null) and
   (there exists a Loop grouping LG such that
      lastNode is not in LG and N is in LG)
  // we did just enter a loop -- but what kind of loop?
  if (looksLikeAForLoop(N, LG))
    loops.push(FOR);
    nextEdge := translateForInit(N, LG);
    emit("begin");
    return(nextEdge, null);
  else if (looksLikeARepeatUntilLoop(N))
    loops.push(REPEAT);
    emit("repeat");
  else
    giveUp(true);

// Translate the node itself.  Note that in the case statement below,
// we can ignore Accept nodes, because they only exist inside STasks,
// and we don't do Stasks.
case (type of N) of:
  Multiassignment Node:  processMultiassignment(N);
  Call Node:             processCall(N);
  Parameter Node:        ;  // ignore it -- we don't do non-strict
                            // procedure calls
  Declaration Node:      ;  // ignore it -- we did declarations earlier
  Epsilon Node:          ;  // ignore it -- empty statement
end;

// Remember that this is the most recent node translated
lastNode := N;

// Pascal has no concept of a Stop node in a procedure
if (N is a Stop node) and
   (N is in a subprogram grouping)
  giveUp();

// if this is a Stop or Return node, there is nowhere to go from here
if (N is a Stop node) or (N is a Return node)
  return (null,null);

// otherwise find the edge and branch to follow outta here
ports := the set of control output ports on N;
validPorts := the empty set;
for each port P in ports:
  branches := the empty set;
  for each destination branch DB in the edge attached to P:
    add DB to branches;
    if (isaSpur(DB))
```

```
        remove DB from branches;
    if (branches is not the empty set)
      add P to validPorts;

  // if there's more than one way out, quit
  if (|validPorts| != 1)
    giveUp(true);
  else
    E := the edge attached to the sole element of validPorts;

    // In theory, we can accept an NPa or an arbitrary NPn edge, as long as
    // exactly one branch is a non-spur, and if we were going to support
    // that, this is where we'd determine whether the edge were valid, and,
    // if so, which branch was the correct one.  However, expediency being
    // everything in this hectic and harried world in which we live, we
    // just reject everything except an NP1 edge.  Support for NPa and NPn
    // edges is built-in to the protocol for the "generate" routine,
    // however, as you'll see if you check out the comment above that
    // routine.
    if (E isn't an NP1 edge)
      giveUp(true);

    if (E has no destination branches) or
       (one or more of E's destination branches terminates at a port that
        is not on a node)
      giveUp(true);
    else
      return (E, null);


// Given a node N that is the first node encountered when entering a Loop
// grouping LG, this routine determines whether the loop in LG should be
// modeled as a Pascal "for" loop.  It returns true only if the node at the
// top of the loop (which may not be the same as N, thanks to loop
// initialization requirements) has both an exit branch and a branch that
// continues on into the loop (not counting branches that go back to the
// top of the loop).
function looksLikeAForLoop(Node N, LoopGrouping LG) return Boolean
  // do a quick check to see if the loop update is consistent with a for loop
  If (there exists an unexecutable edge UE originating on LG and
      having an annotation named "Loop Update") and
     (UE terminates at a Multiassignment node MN) and
     (there exists an assignment A in MN such that A is not of
      the form "VID := VID + 1" or "VID := VID - 1")
    return false;

  // find the node that marks the top of the loop part of the
  // looping construct
  loopTop := findLoopTop(N);
  if (loopTop has more than one control output port P) or
     (there is no edge E attached to P)
    giveUp(true);
```

```
  if (E has 0 or 1 destination branches)
    return false;

  if (E has a destination branch with an annotation named "Loop Exit")
    if (E has exactly two destination branches) and
       (E has a destination branch with an annotation named "Loop Return")
      return false;  // this is preferably modeled as a repeat-until loop
    else
      return true;
  else
    return false;


// Given a node N that is the first node encountered when entering a Loop
// grouping LG, this routine determines whether the loop in LG should be
// modeled as a Pascal "repeat-until" loop.
function looksLikeARepeatUntilLoop(Node N) return Boolean
  // find the node that marks the top of the loop part of the
  // looping construct
  loopTop := findLoopTop(N);

  // for this to be a repeat_until loop, exactly one of the loop return
  // branches leading to loopTop must have as its sole sibling branch a
  // branch that is a loop exit branch.  Return true only if this is the
  // case.
  P := loopTop's control input port;
  branches := all the branches connected to P that have an annotation named
              "Loop Return";
  loopEnds := 0;
  for each branch B in branches:
    E := the edge containing B;
    if (E has a branch with an annotation named "Loop Exit")
      loopEnds++;

  if (loopEnds == 0)
    return false;
  if (loopEnds == 1)
    return true;
  else
    giveUp(true);


// This routine follows the straight-line flow of control starting at N
// until it finds a node that is the terminus of a Loop Return branch.  It
// then returns that node.  If it finds a branch in the flow of control
// before it finds the node it is looking for, it calls giveUp.
function findLoopTop(Node N) return Node
  while (N has no port to which a destination branch with an annotation
         named "Loop Return" is attached) do:
    if (N has more than one control output port P) or
       (there is no edge E attached to P) or
```

```
       (E is not a simple edge)
      giveUp(true);
    else
      N := the node containing the port at which E terminates;

  // if we see the bottom of the loop before seeing the top, quit
  if (N has no input port to which a destination branch with an annotation
      named "Loop Return" is attached)
    giveUp(true);

  // as long as we're in the neighborhood, we might as well check to
  // ensure that no more than one return branch comes here, because Pascal
  // doesn't support multiple loop return points.
  P := N's control input port;
  if (there is more than one branch attached to P with an annotation
      named "Loop Return")
    giveUp(true);
  else
    return N;


// This routine translates the top part of a known for loop, i.e, the part
// "for <variable> := <expression> [down]to <expression> do".  It returns
// the edge to be translated next, i.e., the one that leads out of the loop
// init and into the body of the loop.
function translateForInit(Node N, LoopGrouping LG) return Edge
  if (N isn't a Multiassignment node) or
     (N's content isn't a single assignment A) or
     (A's target isn't a VID)
    giveUp(true);
  else
    theVID := the VID that is the target of A;

  emit("for");
  emit(the name of the variable corresponding to theVID, ":=");

  RHS := the expression that is the right-hand-side of A;
  if (RHS contains references to ports) or
     (RHS contains references to "INPUT")
    giveUp();
  else
    emit(RHS translated into Pascal syntax, i.e., VID's are
         translated into the names of the corresponding variables, etc.);

  UE := the unexecutable edge originating on LG and having an annotation
        named "Loop Update";
  updateNode := the Multiassignment node led to be UE;
  if (updateNode has the form "VID := VID + 1")
    emit("to");
    direction := up;
  else
    emit("downto");
```

```
    direction := down;

  // all we need to do now is grab the value that indicates the loop should
  // terminate
  loopTop := findLoopTop(N);
  P := loopTop's control output port;
  exitBranch := the destination branch on P that has an annotation named
                "Loop Exit";
  C := the condition on exitBranch;
  if (C makes reference to any ports)
    giveUp();
  else if (C != "default")
    if ((direction == up) and
        (C does not have the form "theVID > <expression>")) or
       ((direction == down) and
        (C does not have the form "theVID < <expression>"))
      giveUp();
    else
      emit(the right-hand-side of C translated into Pascal syntax);
  else // C == "default"
    otherBranch := the destination branch on P that does not have an
                   annotation named "Loop Exit";
    C := the condition on otherBranch;
    if (C makes reference to any ports)
      giveUp();
    else
      if ((direction == up) and
          (C does not have the form "theVID <= <expression>")) or
         ((direction == down) and
          (C does not have the form "theVID >= <expression>"))
        giveUp();
      else
        emit(the right-hand-side of C translated into Pascal syntax);

  // finally emit the word "do"
  emit("do");

  return(the edge connected to P);


// This function just says whether it looks like the given branch is a spur
// of control -- i.e., a flow of control that leads off the main path.  A
// spur leads to a Multiassignment node that performs computation only for
// the purpose of shipping data values off to branch conditions.
function isaSpur(DestinationBranch DB) return Boolean
  looksLikeASpur := true;
  if (the node N to which DB leads is a Multiassignment node) and
     (N has at least one data output port) and
     (N has zero control output ports)
    for each output port P on N:
      if (there is an edge E connected to P) and
         (E has exactly one source branch) and
```

```
        (E has at least one destination branch) and
        (none of E's destination branches has a condition) and
        (each of E's destination branches terminates at a branch  condition)
       // do nothing -- these are all requisite conditions for a spur
      else
        looksLikeASpur := false;

    // one last test: E must be an NPa edge or an NP1 edge with exactly one
    // destination branch
    if (looksLikeASpur)
      if ((E is an NP1 edge) and (E has exactly one destination branch)) or
         (E is an NPa edge)
        // do nothing -- all is well
      else
        looksLikeASpur := false;

  else
    looksLikeASpur := false;

  return looksLikeASpur;


// This routine generates code for each of the individual assignments in a
// multiassignemnt.  The order in which the assignments are considered is
// unspecified.  Most of the work in this routine is devoted to handling
// input and output.
procedure processMultiassignment(Node N)
  for each individual assignment A in N:
    LHS := the left-hand side of A;
    RHS := the right-hand side of A;

    // a mainstream Multiassignment node is not supposed to make reference
    // to ports
    if (A contains a reference to a port)
      giveUp();
      return;

    // we could handle this if we were willing to introduce new variables,
    // but we're not, so we don't
    if (LHS == "OUTPUT") and (RHS == "INPUT")
      giveUp();
      return;

    // handle output statements
    if (LHS == "OUTPUT)
      if (RHS is a quoted string)
        newlines := 0;
        while (RHS ends with "\n")
          newlines++;
          RHS := RHS with the trailing "\n" stripped off;

        if (newlines > 0)
```

```
          if (RHS == "")
            emit("writeln;");
          else
            emit("writeln('", RHS, "');");

        for i := 1 to newlines do
          emit("writeln;");

      else // RHS is not a quoted string
        emit("write('");
        emit(the Pascal eqivalent of RHS, e.g., VID's are translated into
             the names of the corresponding variables, etc.);
        emit(");");

    // handle input statements
    else if (RHS == "INPUT")
      emit("read(");
      emit(the VID that LHS must consist of);
      emit(");");

    // handle everything else
    else
      emit(the Pascal eqivalent of A, e.g., VID's are translated into
           the names of the corresponding variables, etc.);


// This routine checks to see if the node N has a valid topology.  If not,
// it issues a diagnostic and forces processing of this routine (or the
// main block) to cease.  Not all nodes are subjected to this filter.  In
// particular, Multiassignment nodes that are used only to compute the
// value of expressions to be used to control token flow are not subjected
// to this.
procedure checkNode(Node N)
  goodNode := true;
  if (N doesn't have exactly one input port IP) or
     (IP is a data port) or
     (N has a data output port)
    goodNode := false;

  // Every node except Stop and Return nodes must have a control output
  // port and must have an edge connected to that port.  The notion of a
  // flow of control just "dying out" doesn't exist in Pascal.
  if (N is not a Stop node) and (N is not a Return node)
    if (N doesn't have a control output port COP) or
       (COP has no edge attached to it)
      giveUp(true);

  // Multiassignment nodes can have data output ports if they're attached to
  // NPa edges that control edge branches, and such nodes are expected to
  // have zero control output ports.
  if (N has a data output port)
    for each data output port DOP on N:
```

```
      if (N is a Multiassignment node) and
         (N has no control output ports) and
         either
           (there is a simple data edge SDE attached to DOP) and
           (SDE's destination branch leads to an input port for a branch
            condition)
         xor
           (there is an NPa edge E attached to DOP) and
           (E has exactly one source branch) and
           (E has at least one destination branch) and
           (none of E's destination branches has a condition) and
           (each of E's destination branches leads to an input port for a
            branch condition)))
        // do nothing -- N, DOP, and E all look legit
      else
        goodNode := false;

  // if this node doesn't look like something to be found in a Pascal
  // program, issue a diagnostic and make a nonlocal jump down the stack to
  // the caller of generate, i.e., the moral equivalent of a longjmp to the
  // point where generate was called.
  if (goodNode == false)
    giveUp(true);


// Generate code for the if/then/else statement(s) corresponding to edge E.
// Each of E's 2 or more destination branches has a condition, and one of
// the edges has the condition "default".
//
// This routine also has to deal with the ends of repeat-until loops.  If
// it finds one, it must emit the code for the body of the loop before
// emitting the code for the exit branch.
//
// It is possible for a branch condition to be dependent on one or more
// data tokens that arrive from one or more Multiassignment nodes.  This
// routine handles that case, but it doesn't handle the case where a value
// is read from standard input and then moved directly into a branch
// condition.  Translating that construct into Pascal is not conceptually
// difficult, but it calls for the introduction of a new variable, and I
// didn't want to support that in this mapping.  It is also worth noting
// that if a Multiassignment node is topologically set up to deliver values
// to branch conditions only (see also the routine translateNode), only
// those assignments in the node that actually produce values that lead to
// branch conditions will be translated.  For example, if such a
// Multiassignment node contains something like this,
//
//    OUTPUT := "Hi Mom!"
//
// no Pascal will be generated for this part of the node.  It would be
// possible to solve this problem (the code could be generated inside the
// routine isaSpur), but I didn't bother.
procedure translateIf(Edge E)
```

```
// check to see if this is really a loop exit
if (E has exactly two destination branches) and
   (one such branch EB has an annotation named "Loop Exit")
  NEB := the destination branch of E that does *not* have an annotation
         named "Loop Exit";
  if (loops.top() == FOR)
    // We're in a for loop.  First generate the loop body code, then
    // generate the code for the exit branch.
    nextNode := the node containing the port at which NEB terminates;
    generate(translateNode(nextNode, NEB));

    nextNode := the node containing the port at which EB terminates;
    loops.pop();
    generate(translateNode(nextNode, EB), false);

  else // we're in a repeat-until loop
    if (NEB doesn't have an annotation named "Loop Return")
      giveUp();
    else
      C := the condition on EB;
      if (C == "default") or
         (C makes reference to any ports)
        giveUp();
      else
        emit("until");
        emit(C translated into Pascal syntax);
        emit(";");

      // The repeat-until is now complete.  Move on to the part of the
      // SPG following the loop.
      nextNode := the node containing the port at which EB terminates;
      loops.pop();
      generate(translateNode(nextNode, EB), false);

else // this isn't the exit point of a loop
  branches := the set of destination branches in E;

  emit("if");

  while (|branches| > 1)
    DB := any element of branches except for the one with the condition
          "default";

    // Translate DB's condition.  This is easier or harder depending on
    // whether data flow is involved.
    if (DB has no data input ports)
      emit(the condition translated into Pascal syntax, i.e., VID's are
           translated into the names of the corresponding variables, etc.);
    else
      // generate the code for the Multiassignment node(s) from which the data
      // tokens must arrive, then generate the code for the if and the
      // condition
```

```
  topologyOK := true;
  for each data input port DIP on the DB:
    if ((there is no simple data edge to DIP from some Multiassignment
        node N) and
        (there is no NPa data edge with a single source branch starting
         at some Multiassignment node N and containing a destination
         branch terminating at DIP)) or
       (there is more than one branch connected to DIP)
      showConfusion();
      topologyOK := false;
      break;

  // if the topology of the condition and its associated
  // Multiassignment nodes was okay, generate code for it
  if (topologyOK)
    for each token T in a left-to-right parsing of the condition on DB:
      if (T is the name of a data input port DIP)
        E := the edge containing the branch attached to DIP;
        P := the port to which E's source branch is attached;
        N := the Multiassignment node containing P;
        if (there is exactly one assignment A in N such that
            P is the target of the assignment)
          RHS := the right-hand side of A;
          if (RHS contains one or more references to data input ports) or
             (EX contains reference to "INPUT")
            showConfusion();
          else
            emit(the Pascal equivalent of RHS, e..g, with VID's
                 translated into the names of the corresponding
                 variables, etc.);

        // more than one assignment is made to P
        else
          showConfusion();

      // T is not the name of a data input port
      else
        emit(the Pascal token eqivalent in meaning to T, e.g., VID's
             are translated into the names of the corresponding
             variables, etc.);

// Okay, we've generated the "if" and the controlling expression.
// Now we can start in on the "then" and the "else" clauses.
emit("then");
emit("begin");
nextNode := the node containing the port at which DB terminates;
generate(translateNode(nextNode, DB));
emit("end");
emit("else");

// we're now done with this branch
remove DB from branches;
```

```
    // okay, only one branch left -- the final "else" clause
    emit("begin");
    lastBranch := the sole remaining element of branches;
    nextNode := the node containing the port at which lastBranch terminates;
    generate(translateNode(nextNode, DB), false);
    emit("end;");


// Generate code for the case statement corresponding to edge E.  E may or
// may not have a branch with the condition "default".  The expression ME
// is the one used to control which branch of the case is taken.
procedure translateCase(Edge E, MultiAssignment_Expression ME)
  branches := the set of destination branches in E such that no branch
              has the condition "default";

  emit("case");
  emit(ME translated into Pascal syntax, e.g., VIDs are translated into
       the names of the corresponding variables, etc.);
  emit("of");

  while (|branches| > 1)
    // choose a branch to translate
    DB := any element of branches that has a condition other than "default";
    branches -= DB;

    // generate the case label
    emit(the value of the constant expression part of the LogicalExp
         that makes up DB's conditon);
    emit(":");

    // and generate the code for the statement corresponding to that value
    emit("begin");
    if (there exists a destination branch in E that has the
        condition "default")
      nextNode := the node containing the port at which DB terminates;
      generate(translateNode(nextNode, DB));
    else
      nextNode := the node containing the port at which DB terminates;
      generate(translateNode(nextNode, DB), false);
    emit("end;");

  // now add a default branch, if necessary
  if (there exists a destination branch DB in E such that DB has the
      condition "default")
    emit("otherwise");
    emit("begin");
    nextNode := the node containing the port at which DB terminates;
    generate(translateNode(nextNode, DB), false);
    emit("end;");

  // finally, add the "end" of the case statement
```

```
  emit("end;");


// This routine does the processing needed to generate Pascal from an SPG
// Call node.  It spends most of its energy actually confirming that the
// Call node can be translated as a procedure call (as opposed to an entry
// call or a function call, neither of which is supported in the subset of
// Pascal this translation supports).  If it finds something of which it
// can make no sense, it generally just quits, but in the case of an entry
// call, it tries to generate a helpful comment.
procedure processCall(Node N)
  entryCall := false;
  if (the content of the first part of N contains an STask specification)
    // this is an entry call -- document it as such
    taskName := "<unknown>";
    if (the value of the expression E for the TID can be statically
        determined) and
       (there is an STask grouping STG in the SPG with a TID of E)
      taskName := nameOf(STG);

    entryName := "<unknown>";
    if (the value of the expression E for the RID can be statically
        determined) and
       (there is an Entry grouping EG in STG with an RID of E)
      entryName := nameOf(EG);

    startComment();
    emit("This is a call to an entry called ", entryName,
         " in a process called ", taskName, ".\n");
    endComment();

  else
    // it's not an entry call, it's a subprogram call
    if (the value of the expression E for the RID in N can be statically
        determined) and
       (there is a Subprogram grouping SG in the SPG with an RID of E) and
       (SG is not inside a Subprogram grouping or STask grouping)
      // do nothing -- the call looks okay
    else
      giveUp();
      return;

    // make the sure the last part of N is as expected -- this rejects, for
    // example, nonstrict calls
    if (the content of the last part of N is other than " := R.r0")
      giveUp();

    // emit the procedure name and a list of actual parameters, if possible
    emit(nameOf(SG));
    if (there is more than one assignment in the second part of N)
      emit("(");
      if (the target of each assignment in the second part of N is of the
```

```
          form "R.px) and
         (the xs form a sequence 0, 1, 2, ... n)
        for each assignment in the second part of N considered in order
          from R.p1 to R.pn:
            emit(the assignment's right-hand side expression translated
                 into Pascal syntax, i.e., VID's are translated into the
                 names of the corresponding variables, etc.);
            if (we haven't emitted the last expression)
              emit(", ");
      else
        showConfusion();
      emit(")");

    // end the procedure call with the obligatory semicolon
    emit(";");


// This routine is called when we can't make any more sense of the SPG.  If
// the flag reallyGiveUp is set, we should do a nonlocal jump down the
// stack to the caller of generate, i.e., the moral equivalent of a
// longjmp.  We do this when processing of the current large block
// (procedure body or main block) should not continue beyond this point.
procedure giveUp(Boolean reallyGiveUp = false;)
  startComment();
  emit("I can't make any more sense of this, so I'm quitting here.");
  endComment();

  if (reallyGiveUp)
    if (generate was called by generateProcedure)
      unwind the stack and continue executing code at the point
      after generateProcedure() calls generate();
    else
      unwind the stack and continue executing code at the point
      after generateMainBlock() calls generate();


// This routine recursively lists the names of the subprograms nested
// inside the specified grouping.  The parameter doComment controls whether
// this routine invocation should start and end the comment containing the
// list of routine names.
procedure listNestedRoutines(Subprogram_Grouping G, Boolean doComment = false)
  if (there exists a Subprogram grouping nested inside G and not
      nested inside any other Subprogram grouping that is inside G)
    if (doComment == true)
      startComment();
    emit("Nested routines (procedure and functions):\n ");
    for each Subprogram grouping SG nested inside G and not
      nested inside any other Subprogram grouping that is inside G):
        emit(nameOf(SG), "\n");
        listNestedRoutines(SG);
    if (doComment == true)
      endComment();
```

```
// This routine generates variable declarations for all the variables
// declared in the specified SPG subgraph.  Note that they are not placed
// in any particular order.  To make the order repeatable in the future, we
// could link them together by adding unexecutable edges annotated with a
// PM-specific annotation to the SPG.  In the interest of brevity, I don't
// do that here.
procedure declareVariables(SPG_Subgraph SPGS)
  for each Declaration node DN in SPGS:
    if (DN is not contained in a Subprogram grouping in SPGS) and
       (DN is not contained in a Parameter Init grouping)
      emit("var ");
      for each variable name VN in DN in the order in which they are listed:
        emit(VN);
        if (VN is the last name in DN)
          emit(": integer;");
        else
          emit(", ");


// This routine emits the name of the specified SPG component.
procedure nameOf(SPG_Component C)
  if (C has an annotation named "Name")
    emit(the value of the annotation);
  else
    emit("<unnamed>");


// This routine emits code show the presence of a funny-looking parameter
// list.
procedure declareConfusingParamList()
  emit("( ");
  showConfusion();
  emit(" )");

// This routine just indicates that something is happening in the SPG that
// can't be expressed in Pascal.
procedure showConfusion()
  startComment();
  emit("???");
  endComment();
```

B.5 Translating an SPG into a Runtime Stack

```
// get ready to run
initialize();

// find out the initial state of SPG execution
for each Scope groupings S in the SPG:
  if (isActive(S))
    addScope(S);

// repeatedly show the state of the stack, then wait for an event that
// might change it
repeat forever:
  // generate the initial display
  showScopes;

  // now wait for event notifications from the SPGM:
  wait for an event notification EN from the SPGM:
  case (EN) of:
    The SPG is about to commence
      preparation for execution:                initialize();
    A token T has just been placed on a port P: processToken(T, P);
    A Subprogram grouping SG has just been
      instantiated:                             processInstantiation(SG);
    A node N has just completed execution:      processNode(N);
    An edge E just completed execution:         processEdge(E);
    otherwise                                   ; // ignore all other events
  end;


// This routine is called when the SPGM indicates that it is about to
// commence preparation for execution.  It may also be called by other
// routines in this section.
procedure initialize()
  activeScopes := the empty set;


// This routine adds S and all Scope groupings containing S to
// activeScopes.
procedure addScope(ScopeGrouping S)
    activeScopes += S;
    for each Scope Grouping SG such that S is contained inside SG:
      activeScopes += SG;


// This routine determines whether S is an active scope.  It does so by
// checking to see if there are any tokens within S.  Subprogram and
// Rendezvous groouping are always considered active, because they might
// have been invoked non-strictly.
function isActive(ScopeGrouping S) return Boolean
  if (S is a Subprogram grouping) or (S is a Rendezvous grouping)
```

```
    return true;

  // any tokens on ports in S?
  for each port P in or on S:
    if (P has a token on it)
      return true;

  // any tokens on edges in S?
  for each edge E in S:
    if (E has a token on it)
      return true;

  // there are no tokens, so the grouping must be inactive
  return false;


// This is a skeletal routine that suggests how the PM's might approach the
// presentation of the information in a stack view.
procedure showScopes()
  for each element S of activeScopes:
    display S in some way, e.g., highlight it, print its name, etc.
    for each Declaration node DN in S:
      for each variable V declared in DN:
        display V's name in some way and V's value in some way;


// If a token arrives on a port, all Scope groupings containing that port
// must be active.
procedure processToken(Token T, port P)
  for each Scope grouping S containing P:
    addScope(SG);


// When a subprogram is instantiated, the new instantiation becomes active,
// unless we're dealing with an SPG generated by one of those sicko
// languages where subprograms aren't scopes.
procedure processInstantiation(SubprogramGrouping SG)
  if (SG is a Scope grouping)
    addScope(SG);


// When a node completes execution, it may cause scopes to become inactive.
// One way is by having a subprogram return, which will occur if the node
// is a return node.  A second way is if the node was the last executing
// entity in a scope, in which case the scope becomes inactive by virtue of
// all flows of computation having died out inside the scope.
procedure processNode(Node N)
  // If N is a return node, remove from activeScopes the executing
  // groupings that are about to go out of existence.
  if (N is a return node)
    G := the unique innermost executing grouping containing N (per
         the discussion of Figure 4-39 in my thesis);
```

```
    for each Scope grouping SG contained inside G:
      activeScopes -= SG;
    if (G is a Scope grouping)
      activeScopes -= G;

  else  // N is not a return node
    if (N has no output ports)
      // if this was the last executing entity in a scope, that scope is no
      // longer active
      for each Scope grouping SG containing N:
        if (isActive(SG) == false)
          activeScopes -= SG;


// When an edge completes execution, it may cause scopes to become
// inactive, because it may carry out the last remaining token from a
// scope.  This routine checks to see if this is in fact the case, and, if
// it is, removes the inactive scopes from activeScopes.
procedure processEdge(E)
  // find the path along which the token that traversed E flowed
  SB := E's source branch along which the token just flowed;
  DBset := the set consisting of E's destination branches along which the
           token just flowed;

  // find the ports involved in the path
  SP := the port to which SB is attached;
  DPset := the maximal set of ports such that each element of the set is
           a port to which an element of DBset is attached;

  // find the scopes that a token might have flowed out of
  scopesToCheck := the empty set;
  for each Scope grouping SG in the SPG:
    if (SP is in or on SG)
      scopesToCheck += SG;

  // find the scopes that a token really did flow out of
  for each element S of scopesToCheck:
    for each element P of DPset:
      if (P is in or on S)
        scopesToCheck -= S;

  // check those scopes to see if they are still active
  for each element S of scopesToCheck:
    if (isActive(S) == false)
      activeScopes -= S;
```

B.6 Modifying the Action Routine for an FSA State

The procedure below resets the action routine for a given FSA state to a named routine. The implementation here takes only a single name, hence the action routine must be a subprogram rather than an STask entry, but adding the necessary code to allow an optional task name is entirely straightforward.

Like the program of Section B.3, the routine below assumes that there is an auxiliary data structure with the following fields for each FSA arc:

- **SPGstart**: The SPG port where the arc originates.
- **SPGend**: The SPG port where the arc terminates.
- **FSAstart**: The FSA state where the arc originates.
- **FSAend**: The FSA state where the arc terminates.
- **values**: The set of value ranges enabling traversal of the arc.
- **subgraph**: The set of components in TSPGS corresponding to this arc.

It further assumes that there is an auxiliary data structure with the following fields for each FSA state:

- **inArcs**: The set of arcs terminating at this state.
- **outArcs**: The set of arcs originating at this state.
- **actionRoutine**: The (TID, RID) pair for the routine to be invoked when this state is entered.
- **initialState**: True iff this state is the FSA's initial state.
- **finalState**: True iff this state is a final state of the FSA.
- **subgraph**: The set of components in TSPGS corresponding to this state.

```
procedure resetActionRoutine(State S, String RoutineName)
  // find the Subprogram grouping corresponding to the routine to call
  routines := the empty set;
  for each Subprogram grouping SG in the SPG:
    if (SG in not inside an STask grouping) and
       (SG is not inside another Subprogram grouping) and
       (SG has an annotation named "Name" with a value of RoutineName)
      routines += SG;

  // disregard any routine that doesn't follow the normal control-flow
  // invocation conventions
  for each member R of routines:
    if (R doesn't have exactly one control InOut port IOP) or
       (IOP doesn't have a PR name of "p0") or
       (R doesn't have exactly one control output port with a PR name of "r0")
```

```
    routines -= R;

// if there are no matching groupings, kvetch and quit
if (|routines| == 0)
  error("No such routine: ", RoutineName);
  return;

// if there is more than one matching grouping, complain about that and quit
if (|routines| > 1)
  error("Routine name is ambiguous: ", RoutineName);
  return;

// all is well, so remember the RID of the routine to call
RID := the RID of the sole element of routines;

// find all the ports in S.subgraph that are endpoints for arcs
// terminating at S
ports := the empty set;
for each arc A in S.inArcs:
  ports += A.SPGend;

// Insert Call nodes into the graph so that a call to the specified
// action routine will be exercised before a token gets to P.  There is a
// special case where insertion of a new Call node is unnecessary: if P
// is on a Call node that is already in S.subgraph and if that Call node
// is already invoking S.actionRoutine, then make the Call node call the
// new action routine.
for each port P in ports:
  // check for the special case
  if (P is on a Call node CN) and
     (CN is in S.subgraph) and
     (the (TID, RID) specification in the first part of CN can be
      statically determined to be equivalent to S.actionRoutine)
    // modify CN so that it calls the new action routine
    set the first part of CN to
      R := the value of RID

  else // no special case -- we have to create a new Call node
    create a new Call node CN;
    newP := a copy of P (including all annotations);
    if (newP has no annotation named "Name")
      give newP an annotation ("Name", "passThroughInput")

    // put ports on CN so we can pass the value arriving at newP on to P
    make newP a port on CN;
    if (P is a control port)
      outPort := a new control output port;
    else
      outPort := a new data output port;
    give outPort an annotation ("Name", "passThroughOutput");
    make outPort a port on CN;
```

```
    // Set up the contents of CN so it calls RID and then passes the
    // token arriving on newP to outPort.  We will thereby effectively
    // pass the token arriving on P's replacement through this node and
    // onto P after the call is complete.
    Give CN the following content:
      R    := the value of RID
      R.p0 :=
           := R.r0, passThroughOutput := the value of newP's Name annotation

    // if P is on a start node in S, move the "Start" annotation from
    // that node to CN
    if (P is on a node N) and
       (N is in S.subgraph) and
       (N is a start node)
      move the annotation named "Start" from N to CN;

    // Make the arcs that used to terminate at P terminate at newP
    for each arc A in S.inArcs;
      if (A.SPGend == P)
        for each destination branch DB in A.subgraph:
          if (DB is attached to P)
            detatch DB from P;
            attach DB to newP;
          A.SPGend := newP;

    // Add an edge from CN's output port to P
    if (P is a control port)
      E := a new control edge with 1 source branch and 1 destination branch;
    else
      E := a new data edge with 1 source branch and 1 destination branch;
    Attach E's source branch to outPort;
    Attach E's destination branch to P;

    // add the new node and edge to S.subgraph
    S.subgraph += CN;
    S.subgraph += E;

// remember the new action routine
S.actionRoutine := (null, RID);
```

Appendix C

SPG-Related Publications

In my research on multiple-view software development, I have tried to address a number of important questions that confront the field. Of these questions, one of the most fundamental is at the same time one of the most neglected, namely, does multiple-view software development offer any measurable advantages compared to traditional software development methodologies? My attempt to address this question was reported in the following paper [137]:

- Scott Meyers and Steven P. Reiss, "An Empirical Study of Multiple-View Software Development," in *Proceedings of SIGSOFT '92: Fifth Symposium on Software Development Environments*, December 1992.

Proceeding on the assumption that multiple-view software development was in fact a desirable goal, I examined different approaches to tool integration in the following article [130], which was an early version of the more extensive review found in Chapter 2:

- Scott Meyers, "Difficulties in Integrating Multiview Development Systems," *IEEE Software*, January 1991.

Early work on the design of SPG-based systems was reported in the first two of the following papers [134, 135]. The third paper focused less on the structural aspects of SPGs and more on the kinds of views they were designed to support and on how they could be used as the core of a development environment [136]:

- Scott Meyers and Steven P. Reiss, "A Semantic Basis for Multiple Views of Programs," in *Advance Working Papers of the Second International Workshop on Computer-Aided Software Engineering (CASE '88)*, July 1988.

- Scott Meyers and Steven P. Reiss, "Representing Programs in Multiparadigm Software Development Environments," in *Proceedings of COMPSAC-89*, September 1989.

- Scott Meyers and Steven P. Reiss, "A System for Multiparadigm Development of Software Systems," in *Proceedings of the Sixth International Workshop on Software Specification and Design*, October 1991.

My decision in early 1989 to develop a prototype SPG implementation in C++ turned out to be a fateful one. The language itself was evolving rapidly [196, 199, 197, 198, 110],

and existing development tools were proving to be unexpectedly frustrating. Steven P. Reiss and I set out to enhance the FIELD environment [165] to offer better support for C++ software development, and we reported our early efforts in this paper [166], which primarily described a class browser we had developed:

- Steven P. Reiss and Scott Meyers, "FIELD Support for C++," in *USENIX C++ Conference Proceedings*, April 1990.

I examined why the tools that were adequate for C programming came up short when applied to C++, and I published the results of my analysis in the following paper [129], which pointed out the impact of such features as dynamic binding and function name overloading:

- Scott Meyers, "Working with Object-Oriented Programs: The View from the Trenches is Not Always Pretty," in *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, September 1990.

At the same time, Moises Lejter and Steven P. Reiss Reiss and I undertook an effort to enhance FIELD and Emacs [192] to specifically address the problems I had identified. Our work was described in this paper [117]:

- Moises Lejter, Scott Meyers, and Steven P. Reiss, "Adding Semantic Information To C++ Development Environments," in *Proceedings of C++ at Work-'90*, September 1990.

These last two papers were subsequently combined and updated and presented as follows [118]:

- Moises Lejter, Scott Meyers, and Steven P. Reiss, "Support for Maintaining Object-Oriented Programs," in *Proceedings of the Conference on Software Maintenance*, October 1991. A slightly revised version was published in *IEEE Transactions on Software Engineering*, December 1992.

Finally, I developed an interest in designing tools that could identify constructs in C++ programs that were "almost always wrong," and Moises Lejter and I presented a list of such constructs and our approach to automatically identifying them in this paper [133]:

- Scott Meyers and Moises Lejter, "Automatic Detection of C++ Programming Errors: Initial Thoughts on a lint++," in *USENIX C++ Conference Proceedings*, April 1991.

I went on to identify many more likely error conditions — some of which were not amenable to automatic detection — in the following book [131]:

- Scott Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley, 1992.

Carolyn K. Duby and Steven P. Reiss and I then collaborated on the design and implementation of a new language that would allow C++ programmers to define custom constraints on their C++ programs and have a system automatically ensure that the constraints were not violated. Our initial reports on this system were published as follows [54, 132]:

- Carolyn K. Duby, Scott Meyers, and Steven P. Reiss, "CCEL: A Metalanguage for C++," in *USENIX C++ Conference Proceedings*, August 1992.

- Scott Meyers, Carolyn K. Duby, and Steven P. Reiss, "Constraining the Structure and Style of Object-Oriented Programs," in *Proceedings of the First Workshop on Principles and Practice of Constraint Programming (PPCP93)*, April 1993.

Bibliography

[1] Gul Agha, "Supporting Multiparadigm Programming on Actor Architectures." Invited Lecture presented at PARLE '89 in Eindhoven, the Netherlands, 1989.

[2] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, *The AWK programming Language*. Addison-Wesley, 1988.

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

[4] Bowen Alpern, Roger Hoover, Barry K. Rosen, Peter F. Sweeney, and F. Kenneth Zadeck, "Keeping Priorities Straight: An Investigation of Incremental Algorithms," Technical Report CS-88-13, Brown University Department of Computer Science, November 1988.

[5] Richard Altmann, Andrew Hawke, and Chris Marlin, "An Integrated Programming Environment Based on Multiple Concurrent Views," *Australian Computer Journal*, vol. 20, no. 2, pp. 65–72, May 1988.

[6] James Ambras and Vicki O'Day, "MicroScope: A Knowledge-Based Programming Environment," *IEEE Software*, pp. 50–58, May 1988.

[7] James Archer, Jr. and Richard Conway, "COPE: A Cooperative Programming Environment," Technical Report TR81-459, Cornell University, June 1981.

[8] Arvind and David E. Culler, *Annual Reviews in Computer Science*, vol. 1, ch. "Dataflow Architectures", pp. 225–253. Annual Reviews, Inc., 1986. Reprinted in [211, 79–101].

[9] Arvind and Rishiyur S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Transactions on Software Engineering*, vol. 39, no. 3, pp. 300–318, March 1990.

[10] E. A. Ashcroft and W. W. Wadge, "Lucid, a Nonprocedural Language with Iteration," *Communications of the ACM*, vol. 20, no. 7, pp. 519–526, July 1977.

[11] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter, "The Pan Language-Based Editing System for Integrated Development Environments," in *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments (SIGSOFT '90)* (Richard N. Taylor, ed.), pp. 77–93, ACM Press, December 1990. Published as *ACM SIGSOFT Software Engineering Notes* 15:6, December 1990.

[12] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein, "The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages," in *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 257–271, June 1990. Published as *ACM SIGPLAN Notices* 25:6, June 1990.

[13] Naser S. Barghouti and Gail E. Kaiser, "Concurrency Control in Advanced Database Applications," *ACM Computing Surveys*, vol. 23, no. 3, pp. 269–317, September 1991.

[14] Dilip K. Barman, "RelType: Relaxed Typing for Intelligent Hypermedia Representations," Technical Report CS-91-26, Brown University Department of Computer Science, April 1991.

[15] J. G. P. Barnes, *Programming in Ada.* International Computer Science Series, Addison-Wesley, 1982.

[16] David R. Barstow, Howard E. Shrobe, and Erik Sandewall, eds., *Interactive Programming Environments.* McGraw-Hill, 1984.

[17] Friedrich Ludwig Bauer, Bernhard Möller, Helmut Partsch, and Peter Pepper, "Formal Program Construction by Transformations – Computer-Aided, Intuition-Guided Programming," *IEEE Transactions on Software Engineering*, vol. 15, no. 2, pp. 165–180, February 1989.

[18] Boris Beizer, *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

[19] L. A. Belady and M. M. Lehman, "A Model of Large Program Development," *IBM Systems Journal*, no. 3, pp. 225–252, 1976.

[20] M. Ben-Ari, *Principles of Concurrent and Distributed Programming.* Prentice Hall International Series in Computer Science, Prentice Hall, 1990.

[21] 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, September 1987. Proceedings of the 1st European Software Engineering Conference, held in Strasbourg, France.

[22] Daniel G. Bobrow and Mark J. Stefik, *The Loops Manual.* Xerox Corporation, Palo Alto, California, 1983.

[23] Grady Booch, *Object Oriented Design with Applications.* The Benjamin/Cummings Series in Ada and Software Engineering, Benjamin/Cummings, 1991.

[24] P. Borras, D. Clément, Th. 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* (Peter Henderson, ed.), pp. 14–24, ACM Press, November 1988. Published as *ACM SIGSOFT Software Engineering Notes* 13:5, November 1988, and *ACM SIGPLAN Notices* 24:2, February 1989.

[25] Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas, "An Overview of PCTE and PCTE+," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Peter Henderson, ed.), pp. 248–257, ACM Press, November 1988. Published as *ACM SIGSOFT Software Engineering Notes* 13:5, November 1988, and *ACM SIGPLAN Notices* 24:2, February 1989.

[26] Gary Bray and David Pokrass, *Understanding Ada.* John Wiley & Sons, 1985.

[27] P. Brinch-Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, pp. 199–207, June 1975. Reprinted in [92, pp. 302–310].

[28] Frederick P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, pp. 10–19, April 1987.

[29] Marc H. Brown, *Algorithm Animation.* Ph.D. thesis, Brown University Department of Computer Science, April 1987.

[30] Marc H. Brown, "Exploring Algorithms using BALSA-II," *IEEE Computer*, vol. 21, no. 5, pp. 14–36, May 1988.

[31] Marc H. Brown, "Zeus: A System for Algorithm Animation and Multi-view Editing," Technical Report 75, DEC Systems Research Center, Palo Alto, California, February 1992.

[32] J. C. Browne, Muhammad Azam, and Stephen Sobek, "CODE: A Unified Approach to Parallel Programming," *IEEE Software*, pp. 10–18, July 1989.

[33] Timothy A. Budd, "Multiparadigm Data Structures in Leda," in *Proceedings of the International Conference on Computer Languages*, April 1992.

[34] M. Cagan, "The HP SoftBench Environment: An Architecture for a New Generation of Software Tools," *Hewlett-Packard Journal*, June 1990.

[35] Brad Campbell and Joseph M. Goodman, "HAM: A General Purpose Hypertext Abstract Machine," *Communications of the ACM*, vol. 31, no. 7, pp. 856–861, July 1988.

[36] "The CASE Interoperability Message Sets: Release 1.0." Published by Digital Equipment Corporation, Silicon Graphics, Inc., and SunSoft (A Sun Microsystems Company), October 1992.

[37] Shi-Kuo Chang, ed., *Principles of Visual Programming Systems.* Englewood Cliffs, New Jersey: Prentice Hall, 1990.

[38] Thomas E. Cheatham, Jr., "An Overview of the Harvard Program Development System," in *Software Engineering Environments* (H. Hunke, ed.), North-Holland, 1981.

[39] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy, "The C Information Abstraction System," *IEEE Transactions on Software Engineering*, vol. 16, no. 3, pp. 325–334, March 1990.

[40] Lori A. Clarke, Debra J. Richardson, and Steven 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* (Peter Henderson, ed.), pp. 153–162, ACM Press, November 1988. Published as *ACM SIGSOFT Software Engineering Notes* 13:5, November 1988, and *ACM SIGPLAN Notices* 24:2, February 1989.

[41] Geoffrey Clemm and Leon Osterweil, "A Mechanism for Environment Integration," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 1–25, January 1990.

[42] CASE Communiqué, "Achieving Agreement." Published by Hewlett-Packard, IBM, Informix, and Control Data Corporation, June 1992. This document is revised on an ongoing basis.

[43] Jeff Conklin, "Hypertext: An Introduction and Survey," *IEEE Computer*, pp. 17–41, September 1987.

[44] Richard Conway, Dan DeJohn, and Steve Worona, "A User's Guide to the COPE Programming Environment," Technical Report TR 84-599, Cornell University Computer Science Department, April 1984.

[45] Doug Cooper, *Standard Pascal.* W. W. Norton & Company, 1983.

[46] René David and Hassane Alla, *Petri Nets & Grafcet: Tools for Modelling Discrete Event Systems.* Prentice Hall International, 1992.

[47] Alan M. Davis, "A Comparison of Techniques for the Specification of External System Behavior," *Communications of the ACM*, vol. 31, no. 9, pp. 1098–1115, September 1988.

[48] Ruth E. Davis, "Logic Programming and Prolog: A Tutorial," *IEEE Software*, September 1985.

[49] Norman M. Delisle, David E. Menicosy, and Mayer D. Schwartz, "Viewing a Programming Environment as a Single Tool," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 49–56, April 1984. Published as *ACM SIGSOFT Software Engineering Notes* 9:3, May 1984, and *ACM SIGPLAN Notices* 19:5, May 1984.

[50] Norman M. Delisle and Mayer Schwartz, "Neptune: A Hypertext System for CAD Applications," in *Proceedings of SIGMOD '86, the International Conference on Management of Data*, pp. 132–139, ACM, May 1986.

[51] Digital Equipment Corporation, Maynard, Massachusetts, *DEC FUSE for ULTRIX*, March 1991. Order number AA-PF4TA-TE.

[52] E. W. Dijkstra, "Cooperating Sequential Processes," Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965. Reprinted in [68, pp. 43–112].

[53] Véronique Donzeau-Gouge, Gérard Huet, Gilles Kahn, and Bernard Lang, "Programming Environments Based on Structured Editors: The MENTOR Experience," in *Interactive Programming Environments* (David R. Barstow, Howard E. Shrobe, and Erik Sandewall, eds.), ch. 7, pp. 128–140, McGraw-Hill, 1984.

[54] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss, "CCEL: A Metalanguage for C++," in *USENIX C++ Conference Proceedings*, August 1992. Also available as Brown University Computer Science Department Technical Report CS-92-51, October 1992.

[55] Mark Edel, "The Tinkertoy Graphical Programming Environment," *IEEE Transactions on Software Engineering*, vol. 14, no. 8, pp. 1110–1115, August 1988.

[56] Mike Carey *et. al*, "The EXODUS Extensible DMBS Project: An Overview," in *Readings in Object-Oriented Databases* (Stanley B. Zdonik and David Maier, eds.), Morgan-Kaufmann, 1989.

[57] *Standard ECMA-149: Portable Common Tool Environment (PCTE) Abstract Specification.* European Computer Manufacturers Association, Geneva, Switzerland, December 1990.

[58] 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*, vol. 9, no. 3, pp. 319–349, July 1987.

[59] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency Handling in Multi-Perspective Specifications." Draft paper of November 18, 1992.

[60] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 1, pp. 31–58, March 1992.

[61] C. N. Fischer, Gregory F. Johnson, Jon Mauney, Anil Pal, and Daniel L. Stock, "The Poe Language-Based Editor Project," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, May 1984.

[62] Gene Fisher, "An Overview of a Graphical Multilanguage Applications Environment," *IEEE Transactions on Software Engineering*, vol. 14, no. 6, pp. 774–786, June 1988.

[63] Joan M. Francioni, Larry Albright, and Jay Alan Jackson, "Debugging Parallel Programs Using Sound," in *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 68–75, May 1991. Published as *ACM SIGPLAN Notices* 26:12, December 1991.

[64] Ferdinando Gallo, Regis Minot, and Iat Thomas, "The Object Management System of PCTE as a Software Engineering Database Management System," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Peter Henderson, ed.), pp. 12–15, December 1986. Published as *ACM SIGPLAN Notices* 22:1, January 1987.

[65] David Garlan, "Views for Tools in Integrated Environments," in *Proceedings of the 1986 International Workshop on Advanced Programming Environments*, 1986.

[66] David Garlan, *Views for Tools in Integrated Environments.* Ph.D. thesis, Carnegie Mellon University, May 1988.

[67] David Garlan and Ehsan Ilias, "Low-Cost, Adaptable Tool Integration Policies for Integrated Environments," in *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments (SIGSOFT '90)* (Richard N. Taylor, ed.), pp. 1–10, ACM Press, December 1990. Published as *SIGSOFT Software Engineering Notes* 15:6, December 1990.

[68] F. Genuys, ed., *Programming Laanguages.* Academic Press, London, 1968.

[69] P. B. Gibbons, "A Stub Generator for Hultilanguage RPC in Heterogeneous Environments," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 77–87, January 1987.

[70] Leonard Gilman and Allen J. Rose, *APL: An Interactive Approach.* John Wiley and Sons, 1976.

[71] Eric J. Golin and Steven P. Reiss, "Representing Visual Programs with Object-Graphs," Technical Report CS-89-05, Brown University Department of Computer Science, February 1989.

[72] Eric J. Golin, Robert V. Rubin, and James Walker II, "The Visual Programmers Workbench," in *Proceedings of IFIP Congress 89*, 1989.

[73] Judith E. Grass and Yih-Farn Chen, "The C++ Information Abstractor," in *USENIX C++ Conference Proceedings*, pp. 265–277, 1990.

[74] William G. Griswold and David Notkin, "Program Restructuring to Aid Software Maintenance," Technical Report 90-08-05, Department of Computer Science and Engineering, University of Washington, September (Revised in December) 1990.

[75] Dirk Grunwald, "A Users Guide to AWESIME: An Object Oriented Parallel Programming and Simulation System," Technical Report CU-CS-552-91, University of Colorado at Boulder Department of Computer Science, November 1991.

[76] Vincent A. Guarna, Jr., Dennis Gannon, David Jablonowski, Allen D. Malony, and Yogesh Gaur, "Faust: An Integrated Environment for Parallel Programming," *IEEE Software*, pp. 20–26, July 1989.

[77] A. Nico Habermann, Charles Krueger, Benjamin Pierce, Barbara Staudt, and John Wenn, "Programming with Views," Technical Report CMU-CS-87-177, Department of Computer Science, Carnegie Mellon University, January 1988.

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

[79] David Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987.

[80] David Harel, "On Visual Formalisms," *Communications of the ACM*, vol. 31, no. 5, pp. 514–530, May 1988. See also followup letters in the December 1988, May 1989, and August 1989 issues of *Comunications of the ACM*.

[81] William Harrison, "RPDE^3: A Framework for Integrating Tool Fragments," *IEEE Software*, pp. 46–56, November 1987.

[82] Mary Jean Harrold and Brian Malloy, "A Unified Interprocedural Program Representation for a Maintenance Environment," in *Proceedings of the Conference on Software Maintenance*, pp. 138–147, October 1991.

[83] *ACM SIGPLAN Notices*, vol. 27, no. 5, May 1992. Special issue on the Haskell programming language.

[84] Roger Hayes, S. W. Manweiler, and Richard D. Schlichting, "A Simple System for Constructing Distributed, Mixed-Language Programs," *Software: Practice and Experience*, vol. 18, no. 7, pp. 641–660, July 1988.

[85] Roger Hayes and Richard D. Schlichting, "Facilitating Mixed Language Programming in Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 12, pp. 1254–1264, December 1987.

[86] Sandra Heiler and Stanley Zdonik, "Object Views: Extending the Vision," in *Proceedings of the Sixth International Conference on Data Engineering*, pp. 86–93, 1990.

[87] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, August 1978. Reprinted in [92, pp. 311–322].

[88] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall International, 1985.

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

[90] J. R. Horgan and D. J. Moore, "Techniques for Improving Language-Based Editors," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, May 1984.

[91] Mark Hornick and Stanley B. Zdonik, "A Shared Segmented Memory System for an Object-Oriented Database," *Transactions on Office Information Systems*, vol. 5, no. 1, January 1987.

[92] Ellis Horowitz, ed., *Programming Languages: A Grand Tour*. Computer Science Press, third ed., 1987.

[93] Susan Horwitz, Jan Prins, and Thomas Reps, "Integrating Non-Interfering Versions of Programs," Technical Report 690, University of Wisconsin-Madison Computer Science Department, March 1987.

[94] Susan Horwitz, Thomas Reps, and David Binkley, "Interprocedural Slicing Using Dependence Graphs," in *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 35–46, June 1988. Published as *ACM SIGPLAN Notices* 23:7, July 1988.

[95] Scott E. Hudson and Roger King, "Semantic Feedback in the Higgens UIMS," *IEEE Transactions on Software Engineering*, vol. 14, no. 8, pp. 1188–1206, August 1988.

[96] *Human-Computer Interaction*, vol. 4, 1989. Special issue on nonspeech audio.

[97] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle, "Fabrik: A Visual Programming Environment," in *Proceedings of the 1988 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '88)*, pp. 176–190, 1988.

[98] Edward A. Ipser, Jr., *Toward a Multi-Formalism Specification Environment*. Ph.D. thesis, University of Southern California, Department of Computer Science, August 1990.

[99] Edward A. Ipser, Jr., David S. Wile, and Dean Jacobs, "A Multi-Formalism Specification Environment," in *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments (SIGSOFT '90)* (Richard N. Taylor, ed.), pp. 94–106, ACM Press, December 1990. Published as *SIGSOFT Software Engineering Notes* 15:6, December 1990.

[100] Suresh Jagannathan and Jim Philbin, "A Customizable Substrate for Concurrent Languages," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 55–67, June 1992.

[101] Douglas W. Jones, "How (Not) to Code a Finite State Machine," *ACM SIGPLAN Notices*, vol. 23, no. 8, pp. 19–22, August 1988. See also Robert A. Lerche's letter to the editor [119] commenting on this article.

[102] M. B. Jones, R. F. Rashid, and M. R. Thompson, "Matchmaker: An Interface Specification Language for Distributed Processing," in *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, January 1985.

[103] Guy L. Steele Jr., *Common Lisp: The Language*. Digital Press, 1984.

[104] Gail E. Kaiser, Simon M. Kaplan, and Josephine Micallef, "Multiuser, Distributed Languge-Based Environments," *IEEE Software*, pp. 58–67, November 1987.

[105] Arthur M. Keller, "The Role of Semantics in Translating View Updates," *IEEE Computer*, pp. 63–73, January 1986.

[106] Brian W. Kernighan and John R. Mashey, "The UNIX Programming Environment," *IEEE Computer*, vol. 14, no. 4, April 1981. Reprinted in [16, pp. 175–197].

[107] Brian W. Kernighan and Rob Pike, *The UNIX Programming Environment.* The Prentice-Hall Software Series, Prentice-Hall, 1984.

[108] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language.* Prentice Hall, first ed., 1978.

[109] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language.* Prentice Hall, second ed., 1988.

[110] Andrew Koenig and Bjarne Stroustrup, "Exception Handling for C++(revised)," in *USENIX C++ Conference Proceedings*, 1990.

[111] Timothy Koschmann and Martha Walton Evens, "Bridging the Gap Between Object-Oriented and Logic Programming," *IEEE Software*, pp. 36–42, July 1988.

[112] Doug Lea, "Steps Toward an Internal Representation System for the Semantic Analysis of C++ Programs." Working paper for OOPSLA OOPDE Workshop., October 1991.

[113] Peter Lee, Frank Pfenning, Gene Rollins, and William Scherlis, "The Ergo Support System: An Integrated Set of Tools for Prototyping Integrated Environments," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Peter Henderson, ed.), pp. 25–34, ACM Press, November 1988. Published as *ACM SIGSOFT Software Engineering Notes* 13:5, November 1988, and *ACM SIGPLAN Notices* 24:2, February 1989.

[114] M. M. Lehman, "Human Thought and Action as an Ingredient of System Behavior," in *The Encyclopedia of Ignorance* (R. Duncan and M. Weston-Smith, eds.), pp. 347–354, London: Pergamon Press, 1977.

[115] M. M. Lehman, "Uncertainty in Computer Application and its Control Through the Engineering of Software," *Journal of Software Maintenance*, vol. 1, no. 1, pp. 3–27, September 1989.

[116] M. M. Lehman, "Uncertainty in Computer Application," *Communications of the ACM*, vol. 33, no. 5, pp. 584–586, May 1990. Technical correspondence.

[117] Moises Lejter, Scott Meyers, and Steven P. Reiss, "Adding Semantic Information To C++ Development Environments," in *Proceedings of C++ at Work - '90*, pp. 103–108, September 1990.

[118] Moises Lejter, Scott Meyers, and Steven P. Reiss, "Support for Maintaining Object-Oriented Programs," *IEEE Transactions on Software Engineering*, vol. 18, no. 12, December 1992. Also available as Brown University Computer Science Department Technical Report CS-91-52, August 1991. An earlier version of this paper appeared in the *Proceedings of the 1991 Conference on Software Maintenance* (CSM '91), October 1991. This paper is largely drawn from two other papers [129, 117].

[119] Robert A. Lerche. Letter to the Editor of *ACM SIGPLAN Notices*, December 1988.

[120] Claus Lewerentz, "Extended Programming in the Large in a Software Development Environment," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Peter Henderson, ed.), pp. 173–182, ACM Press, November 1988. Published as *ACM SIGSOFT Software Engineering Notes* 13:5, November 1988, and *ACM SIGPLAN Notices* 24:2, February 1989.

[121] Mark A. Linton, "Implementing Relational Views of Programs," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Peter Henderson, ed.), pp. 132–140, April 1984. Published as *ACM SIGSOFT Software Engineering Notes* 9:3, May 1984, and *ACM SIGPLAN Notices* 19:5, May 1984.

[122] Barbara Liskov and John Guttag, *Abstraction and Specification in Program Development*. The MIT Press, 1986.

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

[124] Mitchell D. Lubars, "The ROSE-2 Strategies for Supporting High-Level Software Design Reuse," Technical Report STP-303-90, Microelectronics and Computer Technology Corporation, September 1990.

[125] Tara Maja Madhyastha, "A Portable System for Data Sonification," Technical Report UIUCDCS-R-92-1761, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1992. M. S. thesis.

[126] M. Maybee and S. D. Sykes, "Q: Towards a Multi-Lingual Interprocess Communications Model," technical report, University of Colorado at Boulder, 1989.

[127] Bertrand Meyer, "From Structured Programming to Object-Oriented Design: The Road to Eiffel," *Structured Programming*, vol. 10, no. 1, pp. 19–39, 1989.

[128] Bertrand Meyer, *Eiffel: The Language*. Prentice-Hall, 1992.

[129] Scott Meyers, "Working with Object-Oriented Programs: The View from the Trenches is Not Always Pretty," in *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pp. 51–65, September 1990.

[130] Scott Meyers, "Difficulties in Integrating Multiview Development Systems," *IEEE Software*, pp. 49–57, January 1991.

[131] Scott Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 1992.

[132] Scott Meyers, Carolyn K. Duby, and Steven P. Reiss, "Constraining the Structure and Style of Object-Oriented Programs," in *Proceedings of the First Workshop on Principles and Practice of Constraint Programming (PPCP93)*, April 1993. Also available as Brown University Computer Science Department Technical Report CS-93-12, April 1993.

[133] Scott Meyers and Moises Lejter, "Automatic Detection of C++ Programming Errors: Initial Thoughts on a lint++," in *USENIX C++ Conference Proceedings*, pp. 29–40, April 1991. Also available as Brown University Computer Science Department Technical Report CS-91-51, August 1991.

[134] Scott Meyers and Steven P. Reiss, "A Semantic Basis for Multiple Views of Programs," in *Advance Working Papers of the Second International Workshop on Computer-Aided Software Engineering (CASE '88)*, July 1988.

[135] Scott Meyers and Steven P. Reiss, "Representing Programs in Multiparadigm Software Development Environments," in *Proceedings of COMPSAC-89*, pp. 420–427, September 1989.

[136] Scott Meyers and Steven P. Reiss, "A System for Multiparadigm Development of Software Systems," in *Proceedings of the Sixth International Workshop on Software Specification and Design*, October 1991. Also available as Brown University Computer Science Department Technical Report No. CS-91-50, August 1991.

[137] Scott Meyers and Steven P. Reiss, "An Empirical Study of Multiple-View Software Development," in *Proceedings of SIGSOFT '92: Fifth Symposium on Software Development Environments*, December 1992.

[138] Brad A. Myers, "The State of the Art in Visual Programming and Program Visualization," Technical Report CMU-CS-88-114, Department of Computer Science, Carnegie Mellon University, February 1988.

[139] Manfred Nagl, "A Software Development Environment Based on Graph Technology," Technical Report 87-3, Lehrstuhl für Informatik III, Rheinisch-Westfälische Technische Hochschule Aachen, 1987.

[140] James M. Neighbors, *Software Construction Using Components*. Ph.D. thesis, University of California at Irvine, 1980.

[141] James M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering*, September 1984.

[142] Celso Niskier, Tom Maibaum, and Daniel 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.

[143] Marian H. Nodine, *Interactions: Multidatabase Support for Planning Applications.* Ph.D. thesis, Brown University Department of Computer Science, May 1993.

[144] Robert L. Nord and Frank Pfenning, "The Ergo Attribute System," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Peter Henderson, ed.), pp. 110–120, ACM Press, November 1988. Published as *ACM SIGSOFT Software Engineering Notes* 13:5, November 1988, and *ACM SIGPLAN Notices* 24:2, February 1989.

[145] Bashnar Nuseibeh and Anthony Finkelstein, "ViewPoints: A Vehicle for Method and Tool Integration," in *Proceedings of the International Workshop on CASE (CASE '92)*, pp. 50–60, July 1992.

[146] Bashar A. Nuseibeh, "Inter-Viewpoint checks." Personal communication, October 1992.

[147] Patrick D. O'Brien, Daniel C. Halbert, and Michael F. Kilian, "The Trellis Programming Environment," in *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pp. 91–102, October 1987.

[148] Kazuhiro Ogata, Satoshi Kurihara, Mikio Inari, and Norihisa Doi, "The Design and Implementation of HoME," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 44–54, June 1992.

[149] Harold L. Ossher, *A New Program Structuring Mechanism Based on Layered Graphs.* Ph.D. thesis, Stanford University Department of Computer Science, January 1985. Available as Stanford University Computer Science Department Technical Report STAN-CS-85-1078, December 1984.

[150] Harold L. Ossher, "Multi-Dimensional Organization and Browsing of Object-Oriented Systems," in *Proceedings of the IEEE 1990 International Conference on Computer Languages*, pp. 128–135, March 1990.

[151] 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*, pp. 177–184, 1984. Published as *ACM SIGSOFT Software Engineering Notes* 9:3, May 1984, and *ACM SIGPLAN Notices* 19:5, May 1984.

[152] M. Tamer Özsu and Patrick Valduriez, "Distributed Database Systems: Where Are We Now?," *IEEE Computer*, pp. 68–78, August 1991.

[153] M. Tamer Özsu and Patrick Valduriez, *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[154] Frank G. Pagan, *Formal Specification of Programming Languages.* Prentice-Hall Software Series, Prentice-Hall, 1981.

[155] James L. Peterson, "Petri Nets," *ACM Computing Surveys*, vol. 9, no. 3, pp. 223–252, September 1977.

[156] Nathan H. Petschenik, "Practical Priorities in System Testing," *IEEE Software*, pp. 18–23, September 1985.

[157] John Placer, "Integrating Destructive Assignment and Lazy Evaluation in the Multi-paradigm Language G-2," *ACM SIGPLAN Notices*, vol. 27, no. 2, pp. 65–74, February 1992.

[158] Michael Platoff and Michael Wagner, "An Integrated Program Representation and Toolkit for the Maintenance of C Programs," in *Proceedings of the 1991 Conference on Software Maintenance*, pp. 129–137, October 1991.

[159] Terrence W. Pratt, *Programming Languages: Design and Implementation*. Prentice-Hall, 1975.

[160] Georg Raeder, "A Survey of Current Graphical Programming Techniques," *IEEE Computer*, pp. 11–25, August 1985.

[161] Wolfgang Reisig, *Petri Nets: An Introduction*. Springer-Verlag, 1985.

[162] Steven P. Reiss, "Generation of Compiler Symbol Processing Mechanisms from Specifications," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 2, pp. 127–163, April 1983.

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

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

[165] Steven P. Reiss, "Connecting Tools using Message Passing in the FIELD Program Development Environment," *IEEE Software*, pp. 57–67, July 1990. Also available as Brown University Computer Science Department Technical Report CS-88-18, "Integration Mechanisms in the FIELD Environment," October 1988.

[166] Steven P. Reiss and Scott Meyers, "FIELD Support for C++," in *USENIX C++ Conference Proceedings*, pp. 293–299, April 1990. Also available as Brown University Computer Science Department Technical Report CS-93-03, February 1993.

[167] Steven P. Reiss, Scott Meyers, and Carolyn Duby, "Using GELO to Visualize Software Systems," in *Proceedings of UIST '89*, October 1989.

[168] 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.

[169] Thomas W. Reps, *Generating Language-Based Environments*. Ph.D. thesis, Cornell University, 1983.

[170] Charles Rich, "A Formal Representation for Plans in the Programmer's Apprentice," in *Proceedings of the Seventh international Joint Conference on Artificial Intelligence (IJCAI-81)*, pp. 1044–1052, August 1981. Reprinted in M. Brodie, J. Mylopoulos,

and J. Schmidt, editors, *On Conceptual Modelling*, pages 239–270, Springer-Verlag, 1984, and in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[171] Charles Rich and Richard C. Waters, "Automatic Programming: Myths and Prospects," *IEEE Computer*, pp. 40–51, August 1988.

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

[173] Charles Rich and Richard C. Waters, *The Programmer's Apprentice.* ACM Press Frontier Series, ACM Press, 1990.

[174] Charles Rich and Linda M. Wills, "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software*, pp. 82–89, January 1990.

[175] Richard A. Demillo *et al.*, *Software Testing And Evaluation.* Benjamin/Cummings Publishing Company, 1987.

[176] Thomas Rodden, Pete Sawyer, and Ian Sommerville, "Interacting with an Active, Integrated Environment," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 76–84, November 1988. Published as *ACM SIGSOFT Software Engineering Notes* 13:5, November 1988, and *ACM SIGPLAN Notices* 24:2, February 1989.

[177] Douglas T. Ross, "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 1, pp. 16–34, January 1977.

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

[179] Robert V. Rubin, James Walker II, and Eric J. Golin, "Early Experience with the Visual Programmer's WorkBench," *IEEE Transactions on Software Engineering*, vol. 16, no. 10, pp. 1107–1121, October 1990.

[180] Barbara G. Ryder and Marvin C. Paull, "Incremental Data-Flow Analysis Algorithms," *ACM Transactions on Programming Languages and Systems*, vol. 10, no. 1, pp. 1–50, January 1988.

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

[182] Robert W. Scheifler and James Gettys, *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD.* Digital Press, 1990.

[183] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg, *Programming With Sets: An Introduction to SETL.* Springer-Verlag, 1986.

[184] John J. Shilling and Peter F. Sweeney, "Three Steps to Views: Extending the Object-Oriented Paradigm," in *Proceedings of the 1989 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '89)*, pp. 353–361, October 1989.

[185] Nan C. Shu, *Visual Programming*. New York, New York: Van Nostrand Reinhold, 1988.

[186] Abraham Silberschatz, James L. Peterson, and Peter B. Galvin, *Operating System Concepts*. Addison Wesley, third ed., 1991.

[187] Andrea H. Skarra, Stanley B. Zdonik, and Steven P. Reiss, "ObServer: An Object Server for an Object-Oriented Database System," Technical Report CS-88-08, Brown University Department of Computer Science, July 1987. Revised and reprinted in K. R. Dittrich, U. Dayal, and A. P. Buchmann, editors, *On Object-Oriented Database Systems*, Springer-Verlag, New York, 1991.

[188] Douglas R. Smith, Gordon B. Kotik, and Stephen J. Westfold, "Research on Knowledge-Based Software Environments at Kestrel Institute," Technical Report KES.U.85.7, Kestrel Institute, July 1985.

[189] Richard Snodgrass and Karen Shannon, "Fine Grained Data Management To Achieve Evolution Resilience in a Software Development Environment," in *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments (SIGSOFT '90)* (Richard N. Taylor, ed.), pp. 144–156, ACM Press, December 1990. Published as *SIGSOFT Software Engineering Notes* 15:6, December 1990.

[190] Alan Snyder, "The Essence of Objects: Concepts and Terms," *IEEE Software*, pp. 31–42, January 1993.

[191] David Socha, Mary L. Bailey, and David Notkin, "Voyeur: Graphical Views of Parallel Programs," technical report, University of Washington Computer Science Department, Seattle, Washington, 1988. Presented at the ACM Workshop on Parallel and Distributed Debugging, May 1988.

[192] Richard M. Stallman, "EMACS: The Extensible, Customizable, Self-Documenting Display Editor," in *Proceedings of the ACM SIGPLAN/SIGOA Symposium on text Manipulation*, pp. 147–156, June 1981. Reprinted in [16, pp. 300–325].

[193] John T. Stasko, *TANGO: A Framework and System for Algorithm Animation*. Ph.D. thesis, Brown University Department of Computer Science, 1989.

[194] John T. Stasko, "TANGO: A Framework and System for Algorithm Animation," *IEEE Computer*, vol. 23, no. 9, pp. 27–39, September 1990.

[195] Mark J. Stefik, Daniel G. Bobrow, and Kenneth M. Kahn, "Integrating Access-Oriented Programming into a Multiparadigm Environment," *IEEE Software*, pp. 10–18, January 1986.

[196] Bjarne Stroustrup, "Multiple Inheritance for C++," in *Proceedings EUUG Spring '87 Conference*, 1987.

[197] Bjarne Stroustrup, "Possible Directions for C++," in *1987 USENIX C++ Papers*, pp. 399–416, 1987.

[198] Bjarne Stroustrup, "Parameterized Types for C++," in *1988 USENIX C++ Conference*, pp. 1–18, 1988.

[199] Bjarne Stroustrup, "Type-Safe Linkage for C++," in *1988 USENIX C++ Conference*, pp. 193–210, 1988.

[200] Kevin J. Sullivan and David Notkin, "Reconciling Environment Integration and Component Independence," in *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments (SIGSOFT '90)* (Richard N. Taylor, ed.), pp. 22–33, ACM Press, December 1990. Published as *SIGSOFT Software Engineering Notes* 15:6, December 1990.

[201] Sun Microsystems, Inc., *Network Programming Guide*, ch. 6: "External Data Representation Standard: Protocol Specification ", pp. 131–146. Sun Microsystems, Inc., March 1990. This protocol has been designated RFC1014 by the ARPA Network Information Center.

[202] Sun Microsystems, Inc., Mountain View, California, *ToolTalk 1.0 Programmer's Guide*, 1991.

[203] Andrew S. Tanenbaum, "A Tutorial on Algol 68," *ACM Computing Surveys*, vol. 8, no. 2, pp. 155–190, June 1976.

[204] Tim Teitelbaum and Thomas Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM*, September 1981. Reprinted in [16, pp. 97–116].

[205] Warren Teitelman, "A Tour through Cedar," *IEEE Software*, April 1984.

[206] Warren Teitelman and Larry Masinter, "The Interlisp Programming Environment," *IEEE Computer*, vol. 14, no. 4, pp. 25–34, April 1981. Reprinted in [16, pp. 83–96].

[207] Takao Tenma, Hideaki Tsubotani, Minoru Tanaka, and Tadao Ichikawa, "A System for Generating Language-Oriented Editors," *IEEE Transactions on Software Engineering*, vol. 14, no. 8, pp. 1098–1109, August 1988.

[208] R. D. Tennent, "The Denotational Semantics of Programming Languages," *Communications of the ACM*, vol. 19, no. 8, pp. 437–453, August 1976.

[209] Robert B. Terwilliger and Roy H. Campbell, "ENCOMPASS: an Environment for the Incremental Development of Software," Technical Report UIUCDCS-R-1296, University of Illinois at Urbana-Champaign, October 1986. Preprint.

[210] Larry Tesler, "The Smalltalk Environment," *BYTE Magazine*, vol. 6, no. 8, pp. 90–147, August 1981. This paper is one of 11 articles on Smalltalk in this issue.

[211] S. S. Thakkar, ed., *Selected Reprints on Dataflow and Reduction Architectures*. IEEE Computer Society Press, 1987.

[212] Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *Computing Surveys*, vol. 14, no. 1, pp. 93–143, March 1982. Reprinted in [211, 4–54].

[213] Jeffrey D. Ullman, *Principles of Database and Knowledge-Base Systems*, vol. 1. Computer Science Press, 1988.

[214] Nancy L. Urbano, "Heroism and Achilles in the Iliad," Master's thesis, Brown University, 1987.

[215] William W. Wadge and Edward A. Ashcroft, *Lucid, the Dataflow Programming Language*. Academic Press, 1985.

[216] Larry Wall and Randal L. Schwartz, *Programming perl*. O'Reilly & Accociates, 1990.

[217] William B. Warren, Jerry Kickenson, and Richard Snodgrass, "A Tutorial Introduction to Using IDL," *ACM SIGPLAN Notices*, vol. 22, no. 11, pp. 18–34, November 1987.

[218] Richard C. Waters, "Program Editors Should Not Abandon Text Oriented Commands," *ACM SIGPLAN Notices*, pp. 39–46, July 1982.

[219] Richard C. Waters, "The Programmer's Apprentice: A Session with KBEmacs," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, pp. 1296–1320, November 1985.

[220] Richard C. Waters, "Program Translation via Abstraction and Reimplementation," *IEEE Transactions on Software Engineering*, vol. 14, no. 8, pp. 1207–1228, August 1988.

[221] David S. Wile, "Integrating Syntaxes and their Associated Semantics." Submitted for publication, November 1991.

[222] Jack C. Wileden, Alexander L. Wolf, William R. Rosenblatt, and Peri L. Tarr, "Specification-Level Interoperability," *Communications of the ACM*, vol. 34, no. 5, pp. 72–87, May 1991.

[223] Patrick Henry Winston and Berthold Klaus Paul Horn, *Lisp*. Addison-Wesley, 1981.

[224] Wuu Yang, Susan Horwitz, and Thomas Reps, "Detecting Program Components with Equivalent Behaviors," Technical Report 840, University of Wisconsin-Madison Computer Sciences Department, April 1989.

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

[226] Pamela Zave, "A Compositional Approach to Multiparadigm Programming," *IEEE Software*, pp. 15–25, September 1989.

[227] Pamela Zave and Michael Jackson, "Conjunction as Composition," *ACM Transactions on Software Engineering and Methodology*, 1993. To appear.

[228] *Office Knowledge Engineering*, Stanley B. Zdonik, ed., vol. 4, no. 1, February 1991. Special issue on non-standard transaction models.