9-1-1993

Constructing Automatic Program Abstractors of Programming Plans

Eng-Siong Tan
Purdue University School of Electrical Engineering

Henry Dietz
Purdue University School of Electrical Engineering

Constructing Automatic Program Abstractors of Programming Plans

Eng-Siong Tan
Henry Dietz

TR-EE 93-32
September 1993

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907-1285

Constructing Automatic Program Abstractors of Programming Plans

Eng-Swng Tan and Henry Dietz

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907
{tane, hankd)@ecn.purdue.edu
(317) 494 3357

Abstract

Program views are useful aids that can help programmers in understanding aspects of existing programs that they may need to extend or modify. Most views describe detailed information about program components or provide abstractions of relationships between components. Programmers, however, often need abstracted views of ***delocalized programming plans*** to understand how various components throughout a program can be interrelated in different ways to achieve specific goals or program effects. This requires the "weaving" together of diverse information sources that current tools do not adequately support.

This paper presents a new methodology for automatically abstracting delocalized programming plans from programs, and describes automatic abstractors for three specific plans: the calling pattern of a set of functions in a program, the program-wide occurrence of a global variable and the modification effects of a call site. We show how these abstractors can extract instances of programming plans, present them as an integrated set of abstract program views, and support programmer interaction with these views. A framework for constructing automatic abstractors is described, that allows abstractors for a class of delocalized programming plans to be easily constructed.

Descriptive Keywords: program understanding, delocalized programming plans, abstracted program views, system constructors, data flow analysis

1. Introduction

> **Understanding is the primary activity ... Software systems have become so large and complex that developers spend far more time trying to read, understand, modify, and adapt documents than they do creating them in the first place. A successful interactive development environment must support understanding by recognizing, exploiting, and making visible complex relationships within and among documents [BaGr90].**

Our ability to support program understanding is fundamental to software engineering. Unless a programming task can be ***completely*** automated, a programmer's direct involvement in that task will be necessary, and his need for relevant program information essential. Since most software development tasks are sufficiently complex that their complete automation is not

currently possible, nor expected in the near future, the need to understand programs pervades all phases of software development. This is true whether a programmer is seeking to debug, review or validate newly developed software, to reuse mature and established software, or to modify and enhance aging software. Hence, any new tool/methodology that can significantly improve on our current ability to make complex program relationships visible to programmers can be expected to have widespread positive impacts on many phases of software development.

Program understanding has been the focus of a growing body of cognitive research and empirical studies of programmer behavior. One dominant result to emerge from this body of work is the notion of **programming plans,** and their use by programmers during program understanding [Adel81, SoEh84, LeSo86, LiPi86]. A programming plan can be seen as a logical and orderly arrangement of program events/components interrelated in a way so as to achieve a specific goal or effect, or to implement a specific design element. Programming plans may be local to a function, such as the summing or counting of array elements, or they may be &local-*ized,* such as the sequence of I/O operations on an external file, or the program-wide usage of a global variable. Empirical studies of programmers strongly suggests that programming plans play a critical role in program understanding. For instance, [Adel81] and [SoEh84] have both studied how novice and expert programmers view and understand programs, and found that, while novices rely primarily on program syntactic structures, the experts tend to see "chunks" of program fragments organized by program function.[1] Similarly, [LeSo86] and [LiPi86] showed that programmers are especially prone to make software maintenance mistakes due to a lack of awareness and understanding of delocalized programming plans. These cognitive results suggest that tools that can automatically abstract such programming plans, and provide direct support for their discovery, study and understanding, can significantly improve on our ability to understand programs.

The main result of this paper is a framework for constructing automatic abstractors of a wide class of delocalized designer-specified programming plans. Abstractors constructed from the framework will be able to automatically extract instances of the specified programming plan from programs, present these plans as an integrated family of abstracted program views and support programmer interactions with these views. In this paper, we will show how abstractors for these three delocalized programming plans can be easily constructed from the framework:

(1) *The Calling Pattern of Functions* (CPF): Many useful plan instances can be derived from the descriptions of the "pattern of invocation sequence" among functions throughout a program. One example of this is when programmers need to understand the sequence in which file I/O operations may be called in the program, for instance, to guard against the common but serious mistake of closing a file before possible read or write operations to that file. Note that the CPF plan is different from call graph views. Call graph views can only describe which functions can be called by function A; the CPF plan, in addition, can describe the ordering of calls.

[1] This difference in chunking ability of novices and experts is apparently universal, and has been observed in disciplines as diverse as chess and go games, as well as electronics and physics problem-solving [Adel81].

(2) ***Program-Wide Occurrences of Global Variables*** (PWOGV): Most program understanding situations require a programmer to understand the usage of global variables in the program. Typically, programmers need to know about a global variable's initialization and final uses (or lack thereof), potential aliases to the variable, as well as the sequence of operations that may be performed on the variable (e.g. sequence of PUSH and POP operations on a stack). Such questions, in turn, require understanding of the global variable's program-wide "pattern of occurrence", that is, the ordering of references to a global variable and its aliases (i.e. uses and definitions). Other views of a variable's data declaration, abstract data type, or next-use and nextdefinition can describe useful structural or localized information about the variable, but they *can* not describe the delocalized information that the PWOGV plan entails.

(3) ***Modification Effects at Call Sites*** (MECS): To discover and understand the modification effects at call sites, programmers need to "suspend" focus on the current function, and shift attention to the callee. This is complicated by the extensive search required (the callee itself may contain numerous call-sites) and the obscure side-effects that can *occur* (e.g. modification of global variables not visible at call sites). Programmers therefore often need to ask what the modification effects at a call-site are, where they can occur, and how they can occur (i.e. the sequence of calls from the call-site to the modification effect). Data flow analysis information such as interprocedural side effects and interprocedural definition-use chains *can* answer the "what" and the "where", but to our knowledge, no program view exists that can describe the "how".

Until now, the delocalized nature of these plans have rendered them largely obscured in the program code, even with today's code browsers and multiple-graphical-view programming environments. We will show, through code scenarios, that our abstractors can make these programming plans directly visible to programmers.

The main idea of the abstractor construction framework is that a wide class of programming plans, including but not restricted to the above three, can be abstracted ***as regular expressions*** of appropriate program components. The regular expression's "alphabet" consists of those program components that are interrelated by diverse relationships to contribute to the effect achieved in the plan. The "composition" among these program components (through the use of the union, sequence and repeat operations) would then describe the "logical and orderly arrangement" of these events in the programming plan. Identifying the alphabet requires the ability to recognize the "web" of control and data dependences, function call relations, interprocedural side effects and variable aliases relevant to the plan; hence, this process is highly plan-specific. In contrast, the abstraction of this alphabet into a regular expression depends only on the program's control structure. It is this ability to separate the plan-specific from the plan-independent concerns that allows our framework to greatly simplify and ease the task of constructing complete abstractor systems of programming plans. To construct an abstractor for a new programming plan using our framework, the designer has only to specify a single routine for identifying plan components ***within a function***.

Being able to automatically abstract programming plans is one thing; helping programmers to "see" the abstracted information is another. Often, programmers face a dilemma when perusing abstracted program views: while they often need to "see" the abstracted information in a way different from the program's source representation, alternate representations such as

graphical program views present them with the additional burden of having to manually integrate information from multiple program representations, and to relate the non-source representations to source statements in the program. The construction framework addresses this dilemma by presenting each abstraction of a programming plan as an integrated family of program views at two abstraction levels — a **regular expression view** and the **abstract control flow (source) view** — and supports programmer understanding of the programming plan through an interactive process of refining information within these program views.

The rest of this paper is organized as follows. Section 2 motivates the abstractor construction framework by presenting concrete examples of its results (what it can do). *After* introducing the regular expression and abstract control flow views used in the plan abstractors, section 2 presents scenarios of the program views generated from the abstractors of the CPF, PWOGV and MECS programming plans. Section 3 describes the program dependence representation and terminology used in this paper. Section 4 presents the abstractor construction framework. Here, we define the representation of the regular expression used, then describe its construction, and finally sketch it use for view generation and refinement. Section 5 shows how the three abstractors presented in section 2 — the CPF, PWOGV and MECS abstractors — are constructed from the framework. The use of regular expressions restricts the work discussed thus far to non-recursive programs. Section 6 discusses how this restriction is handled by the framework. Section 7 compares this work with related work, and section 8 closes with a summary of this work.

2. Refinabie Program Views: An Introduction and Scenarios for CPF, PWOGV and MECS

In this section, we motivate the plan abstractors by first describing their results, namely, the refinable program views that it generates and the programmer interaction it can support. We first introduce the refinable program views in general, then present examples of these refinable program views generated from abstractors of the CPF, PWOGV and MECS programming plans.

2.1. An Introduction to the Regular Expression and Abstract Control Flow Views

Consider the simple program shown in Figure 1. As a pedagogical example, suppose we are interested in an arbitrary "programming plan", namely, the order in which the statements `stat1`, `stat4`, `stat6`, and `stat8` are executed in the program. Even for this trivial example, the amount of unnecessary details that one needs to sift through in order to answer this question is not trivial. It is easy to see that, in general, it is difficult for programmers to sift through details to "reconstruct" abstract delocalized information. A better approach is to present programmers first with abstracted view of the delocalized information, then allow them to gradually relate the abstracted view to the program code. In this case, the desired execution order can be abstracted by the regular expression

```
(stat1),((stat6,(stat8+ ))* + stat4)
```

The questions we are concerned with here is this: How do we help programmers to "see" this regular expression, and how can we aid programmers in relating this abstracted information to the

program, gradually and incrementally, and under programmer control?

```
program ()                FUNC1 ()                FUNC2 ()
{                         {                       {
   stat1;                    stat5;                  stat7;
   stat2;                    while (pred2) {         stat8;
   if (pred1) {                 stat6;               if (pred4)
      FUNC1 ();                 if (pred3)              stat9;
      stat3;                       FUNC2 ();        }
   }                         }
   else                   }
      stat4;
}
```

Figure 1. A simple program.

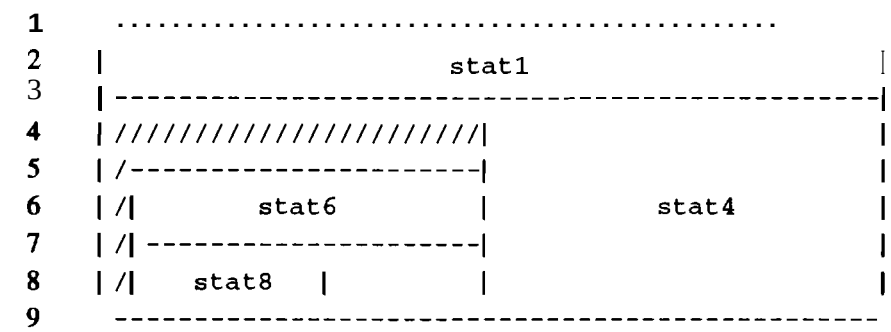

Figure 2. A regular expression view of the plan `statl,((stat6,( + stat8))• + stat4)` in the **program of** Figure **1.**

To help programmers "see" the regular expression, we define a graphical box view patterned after the **Nassi-Shneiderman** Chart **[NaSh73],** called the regular expression view. Figure 2 shows the regular expression view of our example programming plan, and Figure 3 explains the box notations used in this view. Figures **3(a)** through **3(c)** shows the notations for the three basic operations of sequence, branching and iteration. Because iteration in structured programs can occur only in **the** context of a `while(test) body`, we find it convenient to create a special notation to represent **the** typical `test,(body,test)*`, as shown in Figure **3(d).** For `while` loops whose `body` or `test` are not relevant, they are denoted by notations of Figures **3(e)** and 3(f) respectively.

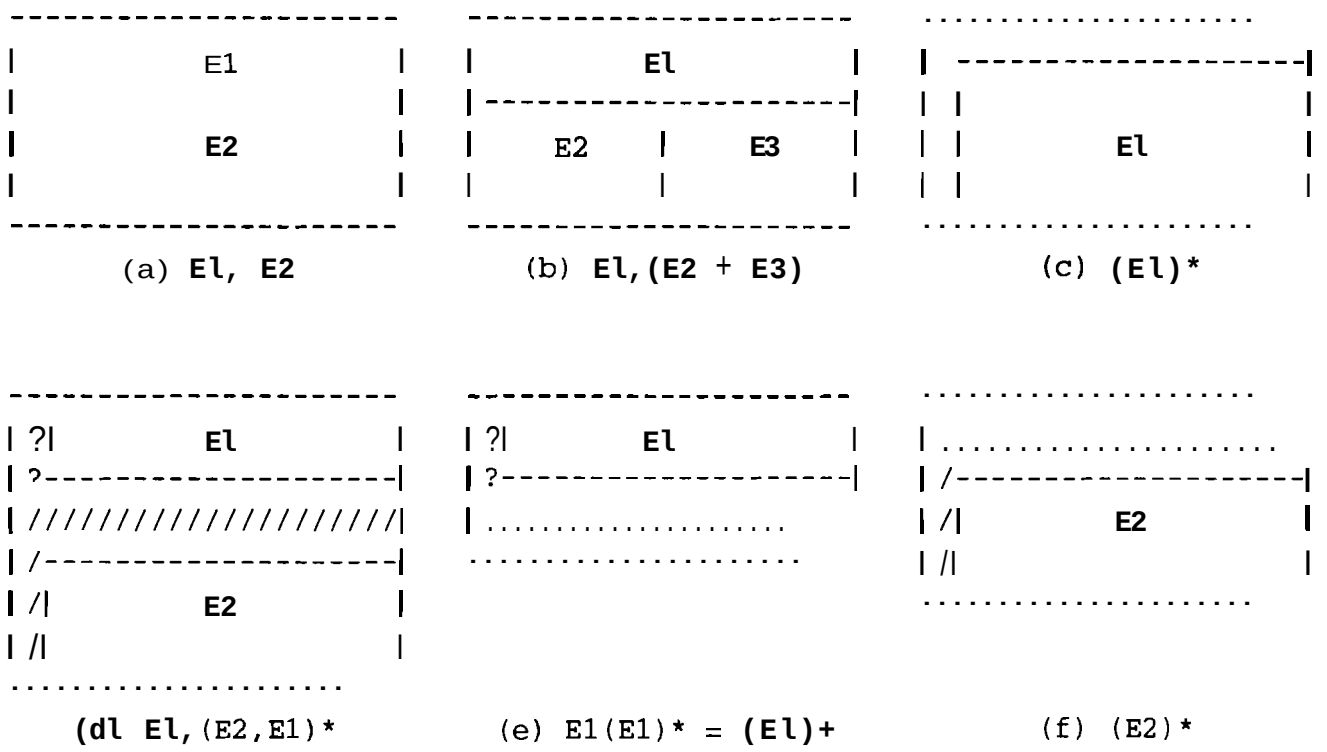

Figure 3. Box notations used in a regular expression view.

While the highly abstract nature of the regular expression view makes it easy to "see" the &localized plan, the view itself cannot describe where and how each sub-expression in the regular expression can occur in the program source. To help programmers do this, the next level of refinable program views provides an **abstract control flow (source) view** of sub-expressions in the regular expression. Selectable sub-expressions are those beginning from the starting symbol to any alphabet symbol, or any branching or iteration operators. For Figure 2, the symbols are `stat1, stat4, stat6` or `stat8`, the branching operators occur on lines 3 and 7, and the only iteration operator is on line 4. Figures 4a, 4b and 4c show the abstract control flow views for the sub-expressions from `stat1` on line 2 up til the branching operator on line 3, the symbol `stat6` on line 6, and finally the entire regular expression, respectively. As these views show, the abstract control flow view is a source representation of the actual program control flow that gave rise to the selected regular sub-expressions. Relevant functions are inlined at call sites, and only the statements corresponding to the symbols, `if` and `while` statements, and calls are included in this view. Hierarchical source elision of block structured constructs is also supported; for instance, in Figure 4c, one can elide the body of the `if (pred3)` statement, or elide the entire function body of **FUNC1.**

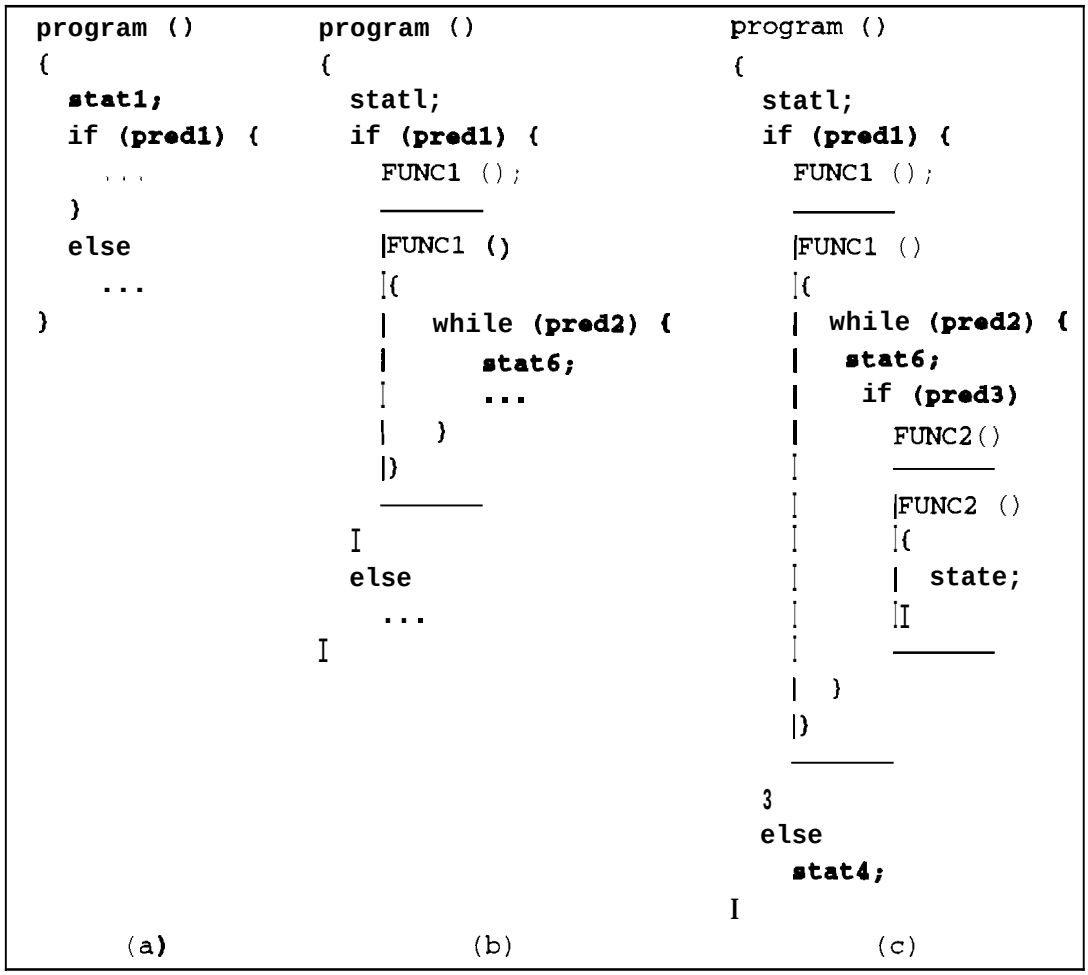

Figure 4. The abstract control flow views for different sub-expressions in Figure 3 from `stat1` to first branching operator on line 3, the symbol `stat6` on line 6, and for the entire regular expression.

2.2. A Scenario of Program Views from the CPF plan abstractor

In this and the next 2 sections, we will show how the above program views can be used to describe three different types of delocalized programming plans. In each case, the structure of the views used are the same; the semantics of views however, will differ.

To illustrate the refinable program views generated by the CPF plan abstractor, this section presents a scenario that describes the sequence of file VO operations in a program. Figure 5 shows the main parts of a postfix evaluation program that reads a list of ';' separated postfix expressions from a file, and appends the evaluated results, in reverse order, back to the file. Assume that `openfile`, `closefile`, `readfile` and `writefile` are special user-defined file I/O functions for the appropriate file, and that all parameters are passed by reference. `PUSH`, POP, ENQ and DEQ are boolean functions implementing LIFO stacks and FIFO queues that return FALSE if bounds are exceeded. Our concern in this scenario is to understand how the

set of file I/O functions, specifically, openfile, closefile, readfile, writefile and endoffile, are called by the main function program.

```
const  MAX 100
global var char postfix [MAX], postfixhead=0, postfixtail=0;   /* a queue */
global var int  result [MAX], resulttop=0;                     /* a stack */
global var int  expr [MAX], exprtop=0;                         /* a stack */
```

```
program()
{
 openfile();
 process();
 closefile();
}

boolean readexpr()
{ boolean ok; char sym;

 ok = readsym(sym);
 while(ok AND sym<>';'){
  ENQ(postfix, sym];
  ok = readsym(sym);
 }
 return(ok);
}

evalsym(int stk[]; char sym)
{
 if(is-operand(syrn))
  PUSH(stk, mkint(sym));
 else
  do-operation(sym);
}
```

```
process()
{
 while(readexpr())
  evalexpr(result);

 writeresult();
}

boolean readsym(char sym)
{
 if(endoffile()){
   closefile();
   return FALSE;
 }
 else{
   sym = readfile();
   return TRUE;
 }
}

do_operation(char sym)
{ int res,opl,op2;

 POP(expr, opl);
 POP(expr, op2);
 res=compute(sym,op1,op2);
 PUSH(expr, res);
}
```

```
writeresult()
{
 while(POP(result, ans))
  writefile(mkchar(ans));
)

evalexpr(int res[])
{
 while(DEQ(postfix, sym))
   evalsym(expr, sym);

 POP(expr, val);
 PUSH(res, val);
}
```

Figure 5. A simple postfix expression evaluation program.

The regular expression view showing the "pattern of invocation sequence" of these file VO functions is shown in Figure 6. This view shows clearly the obvious possibility of the premature closing of the file at not one but two locations (lines 6 and **12** in Figure **6).** One can also "see" that the file will either be closed prematurely, or read at least once (at line 6), but may never be written to. And even if **the** file is not written to at all, the premature closing of the file (at lines 6 or **12)** is still problematic because it will cause **the** closing of a non-existing file at line **20.** Without meticulous reading of the code, **delocalized** non-structural information such as the above is not obvious from **the** program.

Figure 6. The regular expression view of the file I/O operations in Figure 5.

Figure 7 presents a series of abstract control Aow views "explaining" different sub-expressions in **Figure 6.** Figure 7a first shows the relevant program source for the initial symbol **`openfile`** at line **2,** figure 7b describes the expression up to the outer **`while`** test at line **14,** and finally, figure **7c** shows the source statements up to the **`closefile`** and **`readfile`** at line **6** within the outer **`while`** test. The abstracting power of the abstract control Aow view is more obvious from this scenario; the effort needed to manually reconstruct the information presented in Figure **7(c)** from the original program of Figure 5, is at best tedious and error-prone.

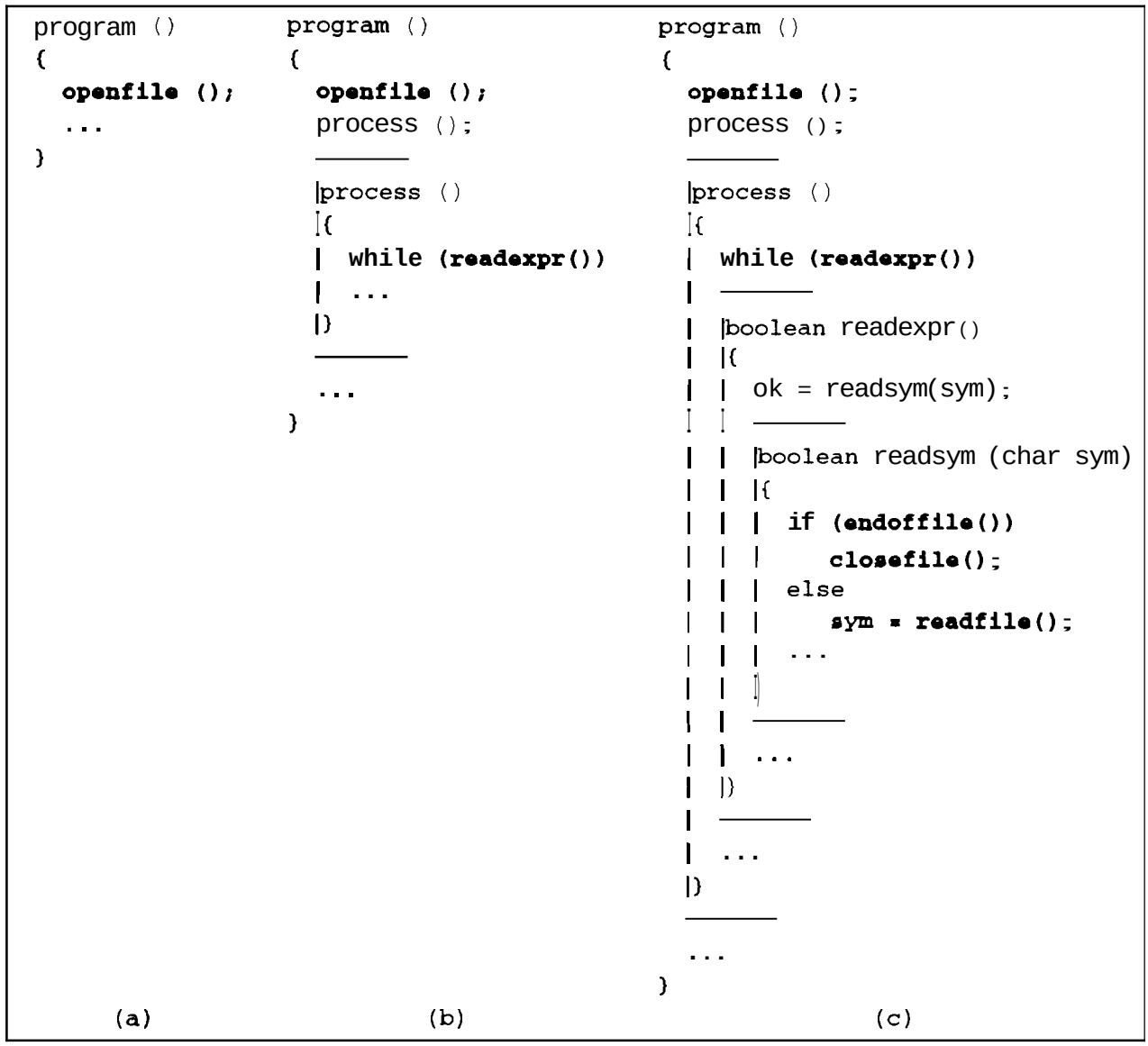

Figure 7. The abstract control flow views showing the program source corresponding to the symbol `openfile` **at line 2 of Figure 6, the** `while` **loop test at line 14 and the symbols** `closefile` **and** `readfile` **at line 6.**

23. A Scenario of Program Views from the PWOGV plan abstractor

The refinable program views can also be applied to understanding the program-wide occurrences of global variables. Here, we present a scenario that describes the sequence of uses and definitions to the global stack variable `expr` in the same postfix evaluation program of Figure 5. In this case, all uses and definitions of `expr` occur in the functions PUSH and POP; hence, we will abstract the global events of interest to just calls to these functions. Note, however, that this is not just another calling pattern of PUSH and POP functions; calls to these

functions made for the `expr` variable must be distinguished from those made for the `result` variable, and aliases to `expr` must be handled.

Figure 8 shows the regular expression view of the program-wide occurrences of `expr`. At least four noteworthy points can be made of this view.

(1) Perhaps the most obvious of these is the characteristic use of the **`expr`** variable as a ***stack*** for postfix evaluation, even though PUSH and the POP-POP-PUSH call sequences occurred in different functions **`evalsym`** and **`do-operation`** in the program.

(2) Calls to PUSH and POP f a the variable **`expr`** are not confused with calls to these same functions made for **`result.`**

(3) The aliasing of **`expr`** to the parameter **`st`**k in the function **`evalsym`** is recognized, and the call to PUSH on **`expr`** here is properly accounted for.

(4) The regular expression view of a global variable's program-wide occurrences provides an unambiguous view of the variable's possible ***initialization*** and ***final uses.***

Through the combination of these features, this regular expression view provides a unique picture of a global variable not before possible, and represents a significant improvement over what current tools can offer.

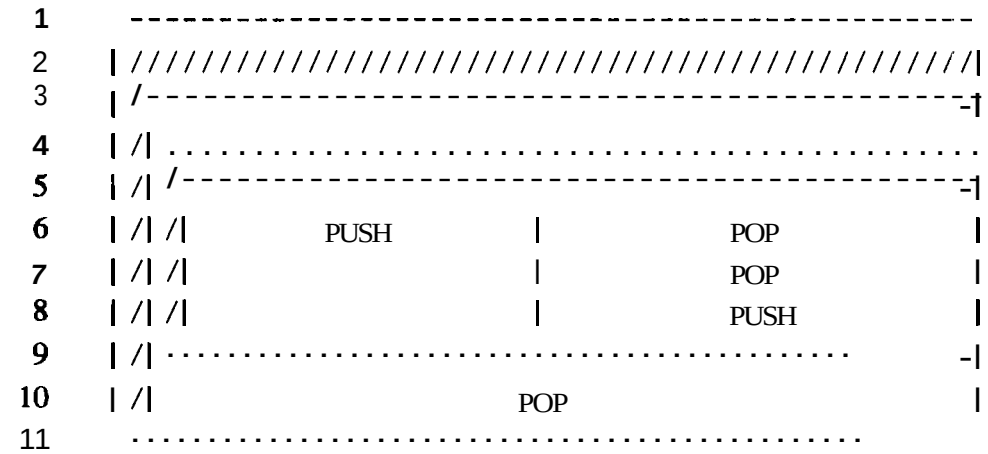

Figure 8. The regular expression view showing the operations on the global stack variable **`expr`** in the program of Figure 5.

Figure 9 shows how information in this unique view can be further refined and improved. Programmers may be curious as to the "reason" for the doubly nested loops on lines 2 and 4 of Figure 8; Figure 9a provides the "explanation". Even with only minimal background information, programmers can infer from this view alone that different postfix expressions may be read in the outer loop, whereas the characteristic PUSH and POP-POP-PUSH is repeated for each postfix expression read. Figure 9b confirms this, by showing the program source for all the uses and definitions of `expr` throughout the entire program.

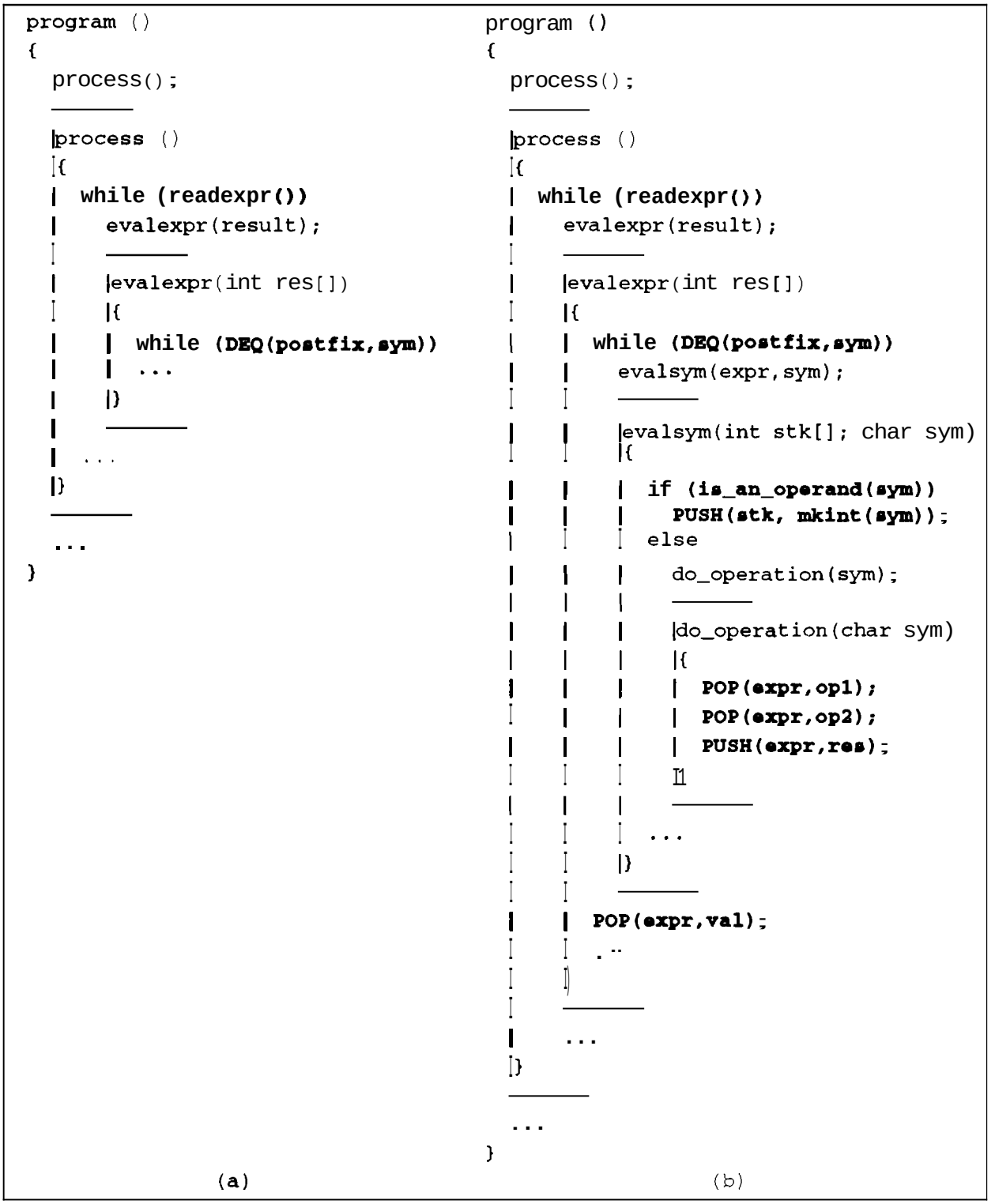

Figure 9. The abstract control flow view f a sub-expressions in Figure 8.

2.4. A Scenario of Program Views from the MECS plan abstractor

A third interesting application for the refinable views is the description of the modification effects at a call site. As mentioned earlier, programmers would not only like to know what variables can be modified and where the definitions can occur, but they also need to understand how the definitions can occur. Consider the program in Figure 10, and the call to function A from program. Interprocedural side effect analysis of this program can tell us that the local variables a and b, and the global variables gvl and gv2 can be modified at this call site. Interprocedural *definition-use* chain analysis can tell us that the possible definition points for the variable a include the definitions to variables m and n in function C and variable t in function D, and that possible definition points for variable b include the definitions to n in function C and t in function D. But what definition sequences are actually feasible, that is, what definitions can occur together on some complete execution of the call to A? Are all variables that can be modified always modified on every execution of A? What are the call sequences from A that doesn't modify, for instance, the global variable gv2?

Figure 11 shows how the regular expression view can answer not just the what and the where, but also the how. In this view, we use the notation "definition@function//variable =" to indicate that the definition in function is a possible definition point of the modification effect variable at the call site. Hence, the n@C//a at line 7 of Figure 11 indicates that the definition of n in function C is a possible definition point for local variable a. In this way, the view clearly describes what the modification effects at the call site are, as well as interprocedural definition-use chain information. But the view describes much more:

(1) The feasible definition sequences can be traced easily. Hence, rather than being presented with a set of possible definitions for all modification effects, the programmer can now trace out sequences of definitions that can actually occur together. For our example, the four possible feasible sequences are easily seen in Figure 11.

(2) Interprocedural side effect information describes what may be modified; Figure 11 allows a programmer to see when a modification effect will or will not occur. For instance, one can easily see in Figure 11 that if the left (TRUE) branch of the branch point at line 3 is taken, the local variable b will not be modified; likewise, taking the right (FALSE) branch implies that the global variable gv2 will not be modified.

(3) Notice that, unlike the PWOGV plan, the variable aliasing pattern described applies to all variables involved at the call site. That is, the aliasing pattern for the MECS plan is call-site-specific, rather than variable-specific.

(4) Often a function may be called several times by another function, each time for different reasons. Figure 11 describes such invocation-specific contributions of a function. For instance, the function C is called twice from the call site, but each call contributes differently to the modification effects seen at the call site in program: when called through the functions A and B, C modifies the global variable gv2 and the local variable a at lines 5 and 7, but when called directly from A, C modifies the local variables a and b at lines 4 and 6.

The abstract control flow view in Figure 12 shows the program source for the ''leftmost'' feasible definition sequence of Figure 11.

```
global var int gvl = 0;
global var int gv2 = 10;

program ()                A (int x,y;)              C (int m,n;)
(   int  a, b;            {                         {
                             gvl = x + 2;              m = m + gvl;
   a = read ();              if ( gvl < gv2 )          if (m > 60)
   A (a, b);                    B (gv1, x);               n = m;
   write (a, b);             else                      else
}                               C (x, y);                 D (n);
                          }                         }


B tint p, q;)             D (int t;)
{                         {
   p = q * q;                t = gvl + gv2;
   C (gv2, q);            }
}
```

Figure 10. A program to illustrate the delocalized plan of a call-site's modification effects.

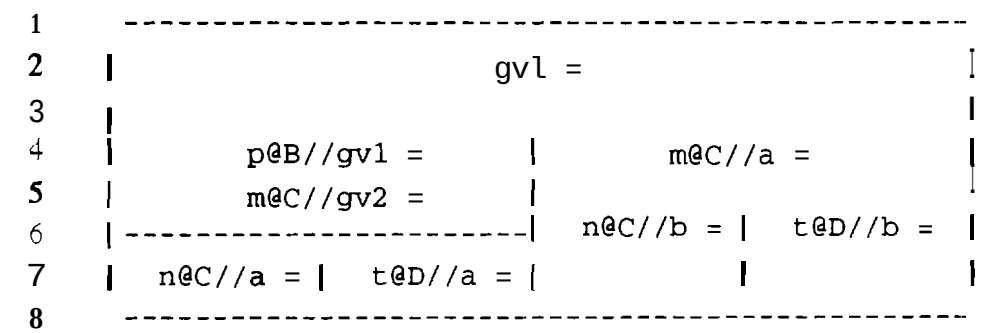

Figure 11. A regular expression view describing the modification effects at a call site.

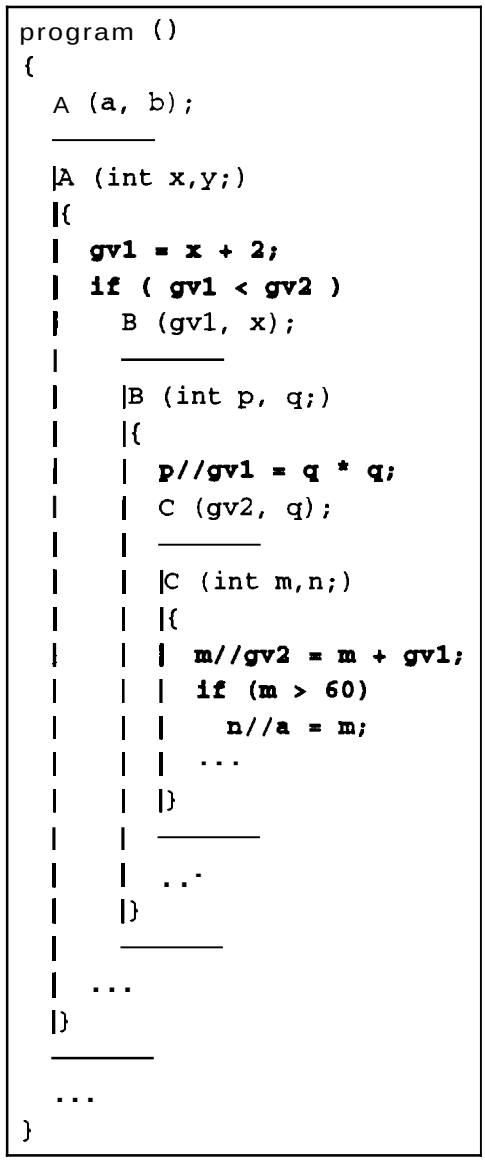

Figure 12. The abstract control flow view of a feasible sequence of modification effects.

3. Program Dependence Terminology and Representation

This section briefly introduces the dependence analysis terminology used in this paper. Throughout this paper, we will assume a language consisting only of assignments, if and while statements, as well as function calls. All call site parameters are assumed to be passed by reference; global variables are permitted, as is variable aliasing through parameter binding. We will, however, assume that no recursion is present; section 6 discusses the problem of recursion, and describes the extensions necessary to handle it.

The program dependence representation is only briefly sketched here; the appendix contains a full definition of the ***program dependence graph*** (PDG) and the ***system dependence graph***

(SDG) used to representation dependences in a program. Our representation scheme corresponds most closely to that used in [HoRe88]: Local data and control dependences in functions are represented in a ***program dependence graph*** (PDG) for each function; these PDGs are then connected by call invocation and parameter-linkage edges to form the ***system dependence graph*** (SDG) representation of the entire program.

Specifically, the PDG representation is a graph with control vertices such as ***seq, if, while, and*** value vertices to represent assignment statements and predicates in control statements. A control dependence edge from vertex v_c (a control vertex) to vertex v is denoted by $v_c \rightarrow_{cd} v$, and may be labeled ***true*** or ***false,*** or it may be unlabeled. A labeled edge (from if and ***while*** vertices) indicates that v will be executed only if v_c is executed ***and*** the predicate associated with it evaluates to a result matching the label; an unlabeled edge (from ***seq*** or program ***entry*** vertices) indicates that v will always be executed if v_c is executed. A data dependence edge from an assignment vertex v_a to vertex v, indicates that a variable is defined at v_a that may be used at v, and an execution path exists from v_a to v ***that*** does not redefine the variable. If this execution path consists of backedges, then the data dependences edge is loop-carried and is denoted by $v_a \rightarrow_{dd_l} v$; otherwise it is loop-independent and is denoted by $v_a \rightarrow_{dd_l} v$.

The SDG representation of a multi-function program is essentially a "supergraph" consisting of the PDGs of each function in the program, extended to represent function calls and parameter passing. These extensions require information about what variables may be used or modified as a result of invoking a function ***Interprocedural flow-insensitive side-effect analysis*** [CoKe88 etc] of the program is required. This analysis annotates every function f in the program with two variable sets $GMOD(f)$ and $GUSE(f)$, each containing all the formal parameters and global variables that may be modified and used, respectively, as a result of f's invocation.[2] Once the GMOD and GUSE sets of functions in the program are known, the set of possible-definitions and possible-uses at a call site c, denoted by $MOD(c)$ and $USE(c)$ respectively, that calls function g can be determined from $GMOD(g)$, $GUSE(g)$ and the parameter bindings at c. The SDG is then constructed from the PDGs and these interprocedural sets in two steps. Firstly, each call site c is linked to its callee function, and secondly, parameter linkage is established by by connecting each variable in USE to its corresponding parameter in GUSE and connecting each variable in GMOD to its corresponding parameter in MOD.

4. A Framework for Constructing Programming Plan Abstractors

We now turn our attention to the construction framework for constructing a class of programming plan abstractors.

As described in the introduction section, the basic actions of an abstractor are to abstract a programming plan as a regular expression, present this abstraction as an integrated family of

[2] *GMOD (f)* and *GUSE (f)* are defined in [CoKe88] to contain *all* variables that may be modified and wed, and may include local variables off. F a our purposes, local variables are irrelevant, and we will restrict *GMOD (f)* and *GUSE (f)* to variables that may be visible outside *f*, namely, the formal parameters and global variables.

refinable program views, and support programmer interaction with these views. Central to these actions is the need for a compact representation of the abstracted information that can not only facilitate the generation of the refinable program views, but can also support programmer interaction with the views, without having to repeat expensive analyses. To this end, we introduce the event flow graph (EFG) representation of the regular expression, a data structure that is specially annotated to meet the needs of view generation and programmer interaction.

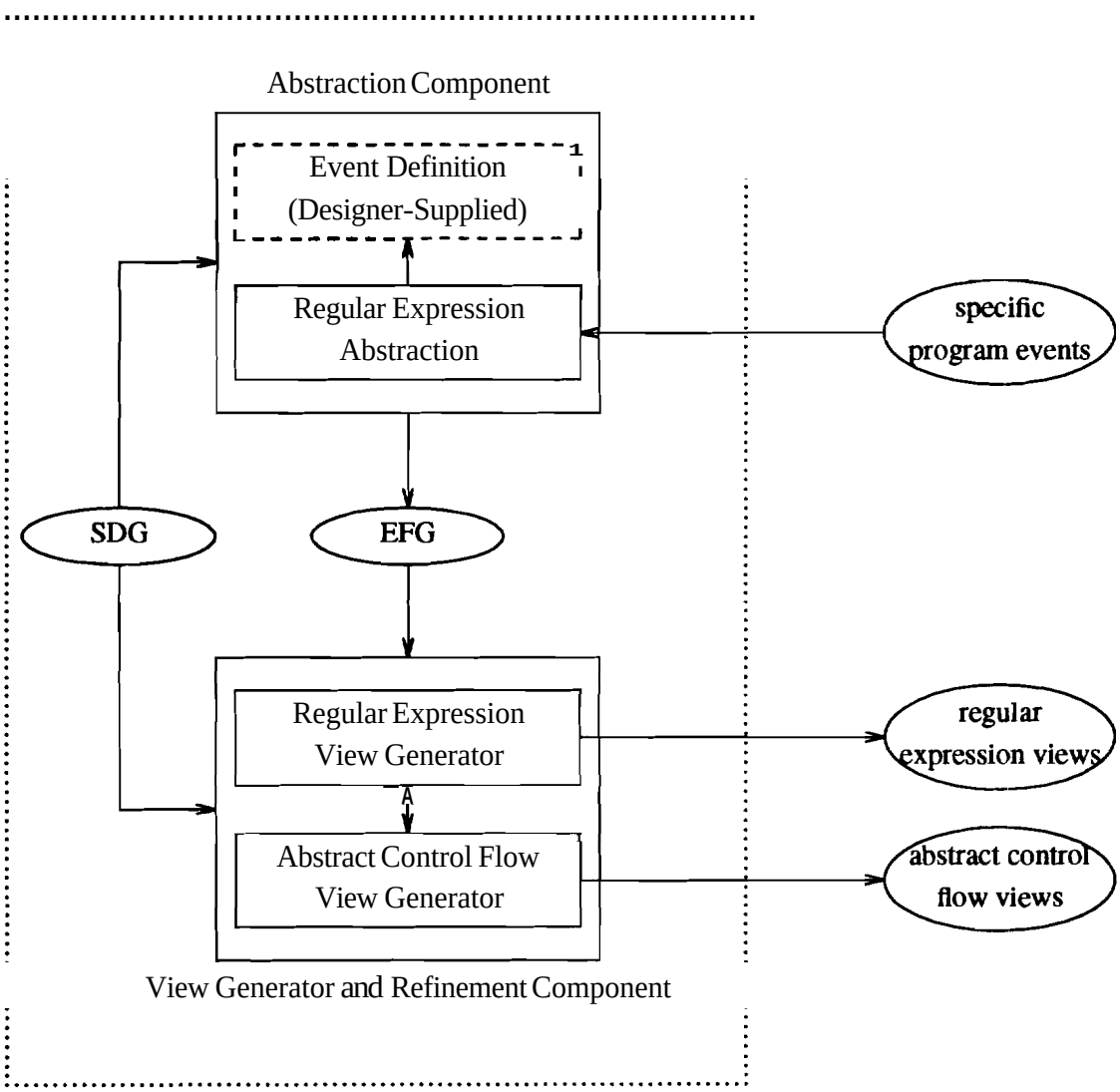

Figure 13. System Architecture of a Programming Plan Abstractor Constructed from the Framework.

Figure 13 shows the system architecture of abstractors constructed from the framework. The figure shows the two main components, the abstraction component and the program view generation and refinement component, and the use of the SDG and EFG representations by these

components. The abstraction component accepts specific program events of the appropriate type (functions, global variables or call sites) from the end-user, and builds an EFG representation of the programming plan abstraction. The program view generator and refinement component generates from this EFG representation the regular expression view, uses the EFG to guide the generation of the abstract control flow view from the SDG, and supports programmer interactions with these views.

Figure 13 also indicates the ease with which programming plan abstractors can be built from the construction framework, and the extent to which the construction framework acts as an "abstractor generator". To build an abstractor system for a new programming plan, the abstraction designer has only to provide the plan event definition sub-component (the dashed box) which, as we will show, amounts to one routine for identifying plan events within the body of a function. All other sub-components in the abstractor (in solid boxes) are programming-plan-independent, and is supplied by the construction framework.

The focus of the rest of this section will be on the EFG data structure and the design of the programming-plan-independent sub-components of Figure 13. The plan-specific sub-component will only be briefly overviewed here, to allow a complete picture of the workings of the construction framework to be painted; the specification this sub-component will be more fully discussed in section 5, where the CPF, PWOGV and MECS specific event definition routines will also be presented. Section 4.1 opens the discussion by motivating and defining the EFG representation. Section 4.2 describes the abstraction component, and presents the EFG construction algorithm for a given programming plan, and overviews the event definition routine it expects the designer to supply. Section 4.3 discusses the view generation and refinement component, and sketches how the refinable program views are generated from the EFG and SDG, and how programmer requests for view refinement are handled. The example from section 2.1 will be used throughout the following discussion.

4.1. The Event Flow Graph (EFG)

We begin the discussion of the construction framework by discussing the motivation behind the data structure used to represent the abstraction. If all that the abstractor had to generate is only the regular expression view, and no support for programmer interaction is required, then the usual abstract syntax tree representation of the regular expression [AhSe86] would have sufficed. However, with the addition of the abstract control flow view, and the need to support programmer refinement of both program views, an adequate representation of the abstraction will have to meet these additional requirements:

(1) ***Allow any sub-expression within the regular expression view to be easily selected, and to generate its corresponding source representation in the abstract control flow (source) view.*** **We wish *to* support programmer refinement of the regular expression view, *so* that any valid sub-expression within the regular expression can be easily related to the program source, but without including statements irrelevant to the understanding of the sub-expression's occurrence in the program.**

(2) ***Support hierarchical elision of the abstract control flow (source) view.*** **We wish to support programmer refinement of the abstract control flow view, so that programmers can hierarchically elide the source representation of structured constructs in the abstract control flow view, such as `if` and `while` control statements, as well as calls to functions.**

The ***event flow graph*** (EFG) is a regular expression representation that meets these requirements. Figure **14** shows the EFG for the pedagogical example given in section **2.1,** corresponding to the regular expression `(stat1),((stat6,(stat8+ ))* + stat4).` (Compare this to the regular expression view and the abstract control flow source view shown in Figures **1** and **2.)**

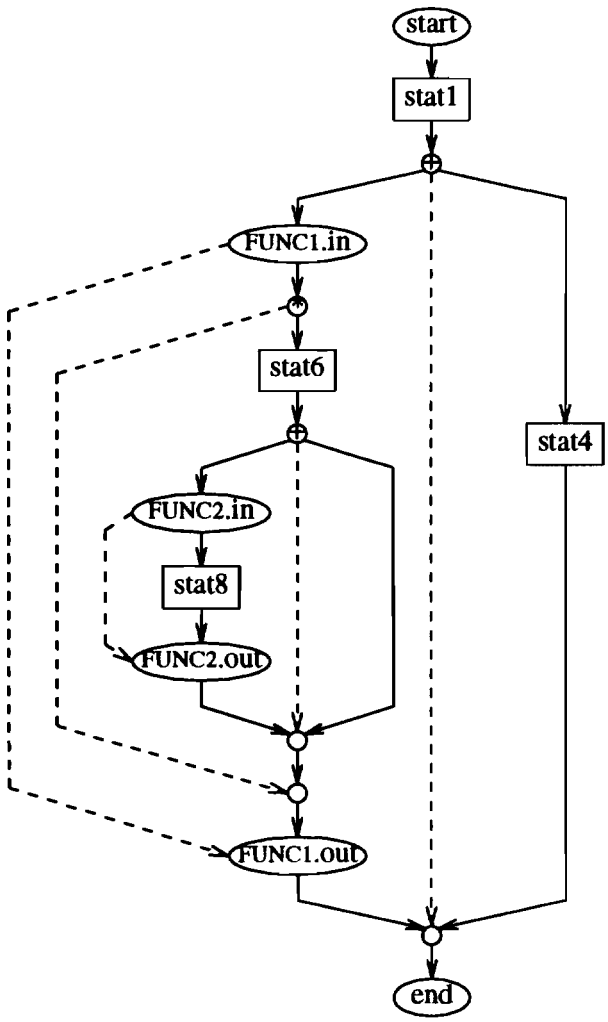

Figure 14. Example EFG for the example of section **2.1.** Boxes indicate ***event*** vertices; circles and ellipses indicate ***control-flow*** vertices. Solid arrows are ***sequence*** edges and dashed arrows indicate ***pairing*** edges.

Formally, *the EFG* is a flow graph, denoted by *EFG,* whose vertex set V_{EFG} consists of *event* vertices (indicated in Figure 14 as boxes) representing the event statements of interest, *as* well *as control-flow* vertices (indicated *as* ellipses and circles) representing any statement needed to describe the flow of control between the event vertices. Control-flow vertices include two distinguished vertices *start* and *end* marking the ends of the entire *EFG,* the "+" and "*" vertices representing relevant `if` and `while` statements, and *func.in* and *func.out* vertices representing entries into, and exits out of, function *func.* Additionally, each EFG vertex "points at" its corresponding vertex in the SDG. The labels of *event* vertices are designer-specifiable. The edge set E_{EFG} consists simply of *sequence* edges (indicated by solid arrows) that connects two adjacent vertices in the regular expression, and the *pairing* edge (indicated by dashed arrows) that marks the beginning and end of control constructs.

43. The Abstraction Component: Constructing the EFG

Figure 14 above suggests that the EFG closely minors the syntactic structure of the program, and hence can be constructed in a syntax-directed manner. The semblance to the syntactic structure, however, is not exact, since only event and "relevant" control-flow statements are to be included. Figure 15 shows the general syntax-directed construction "patterns" employed in the EFG's construction; the parentheses indicates the possibility of empty expressions resulting from absence of event/relevant statements. In this section, we first overview the `IdentifyEvent` routine that the designer needs to provide to identify the event and relevant statements for the EFG construction process, then present the `BuildEFG` algorithm that implements the construction patterns of Figure 15.

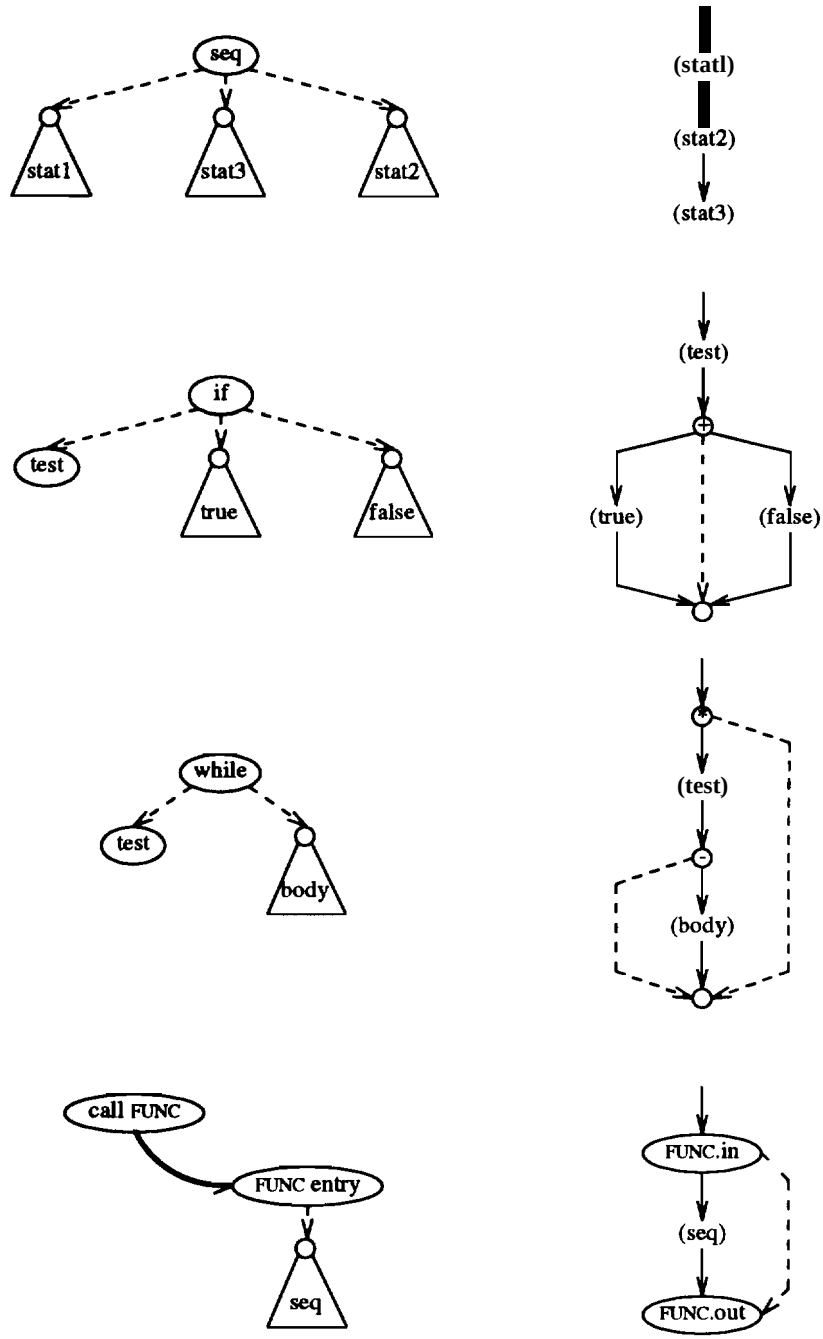

Figure 15. Basic syntax-directed patterns for constructing the EFG (right hand structures) from the SDG (left hand structures).

4.2.1. The Designer-Supplied `IdentifyEvent` Routine: An Overview

An integral part of the EFG construction algorithm involves determining which statement in a program constitutes an event or a relevant statement in a programming plan. This part is clearly programming-plan-dependent, as opposed to the rest of the algorithm that implements the plan-independent syntax-directed composition patterns. An important aspect of our EFG construction algorithm is the separation of this programming-plan-specific event identification from the plan-independent EFG construction. This separation of concerns in fact simplifies the event identification process, so that an abstraction designer only needs to specify a single routine, to be called `IdentifyEvent`, that identifies the event and relevant statements *within a function*, rather than for the entire program. The work of constructing an entire programming plan abstractor is thereby reduced to the specification of a single routine.

```
procedure IdentifyEvent(entry, cc)
input
  entry: a function entry vertex of function f in SDG
  cc: the calling context
begin
  for each vertex v in f do
    if v ∈ {stat1, stat4, stat6, stat8} then
      mark v as an event
      set v.printlabel to text of v
    else if v is a call site then
      mark v as a relevant call site
      set v.calling_context to ∅
    endif
  endfor
end
```

Figure 16. `IdentifyEvent` routine for the example from section 2.1.

The EFG construction algorithm expects this `IdentifyEvent` to be a two-argument procedure that is called on a function f, and whose main purpose is to identify and mark the event statements and relevant call sites within f. The first argument is the *function entry* vertex of the function's PDG_f, from which statements within f can be accessed. The second is an argument for carrying *calling context* information from call sites to the function f, since local information alone is generally not sufficent for identifying event and relevant statements, and information about the context of the specific call site (such as variable aliases) is often required. Figure 11 gives the `IdentifyEvent` routine for constructing the programming plan desired in the pedagogical example.

The event identification process itself is highly plan-specific, and may require information about call relations, control or data dependences, interprocedural summary side effects or variable aliases, or any combination of these. In section 5, we will show complete examples of the `IdentifyEvent` routine for the CPF, PWOGV and MECS programming plans. Here, to understand the role of `IdentifyEvent` in the EFG construction algorithm, it is only

necessary to specify the obligations that the EFG algorithm expects this routine to fulfil. `IdentifyEvent` must

(1) mark all non-call-site statements in PDG_f as either an "event" or a "non-event",

(2) mark all call sites in PDG_f as either an "event", a "relevant" call site, or a "non-event",

(3) specify the labeling of *EFG event* vertices, by annotating *the printlabel* field of *the* "event" statements in PDG_f, *and*

(4) compute and note the new calling context of all "event" and "relevant" call sites in PDG_f, by annotating their *calling–context* fields with the calling context specific to that call site.

In obligation (2), "relevant" call sites are those that may (transitively) call other functions containing event statements, but are not themselves events in the programming plan (and hence should not appear in the regular expression view).

43.2. The EFG Construction Algorithm: The `BuildEFG` Routine

We are now ready to present the EFG construction algorithm. Figure 17 shows how the process is initiated. Three basic actions are required: specify the SDG vertex to begin the plan abstraction, initialize calling context (if applicable), then make the first call to the main EFG construction routine, `BuildEFG`. For our pdeagogical example, we simply set *sdg* to the function *entry* vertex for `program`, and initialize *cc* to ∅.

```
procedure GenericAbstractor()
declare
  sdg: the plan-specific starting vertex in SDG
  cc: the plan-specific initial calling context
  start: the EFG start vertex
  end: a pointer to EFG vertices; will point at end vertex on return

  start = mknode("start")
  call BuildEFG(end=&start, sdg, cc)
end
```

Figure 17. Starting the EFG construction process.

The main routine in the algorithm is `BuildEFG`, presented in Figure 18, that walks the SDG's control subgraph recursively (the subgraph induced by all the SDG vertices and the union of all the control dependence edges). `BuildEFG` is called with a pointer to the current last vertex of the partially constructed EFG (where the next EFG vertex is to be added), the current SDG vertex and the calling context needed for `IdentifyEvent`. It returns with the first argument pointing at the new last vertex of the grown EFG. On entry to a function, a call to the `IdentifyEvent` routine is made for that function, before any EFG construction is continued. Construction of the EFG then proceeds as per the syntax-directed patterns of Figure 15. `MarkRelevant` is called after `IdentifyEvent` to avoid having to process empty `if` or `while` statements. `GrowEFG` takes a pointer to the current last vertex of the EFG, and a new EFG vertex, and makes the new vertex the new last vertex, with the pointer updated accordingly.

```
procedure BuildEFG(efg, sdg, cc)
input
  efg: pointer to the current last vertex of EFG before call
  sdg: current vertex in SDG
  cc: the calling context
output
  efg: pointer to new last vertex of EFG before returning from call
declare
  R, P, Q: EFG vertices
  efgT, efgF: pointers to EFG vertices
begin
  if sdg is a FUNCTION–ENTRY node then
    call IdentifyEvent(sdg, cc)   /* calls Designer-supplied routine to define events in the function */
    call MarkRelevant (sdg)
    call BuildEFG(efg, sdg.seq, cc)
  else if sdg is a relevant SEQ node then
    for each child of sdg, in order, do
      call BuildEFG(efg, child, cc)
    endfor
  else if sdg is a relevant IF node then
    call BuildEFG(efg, sdg.test, cc)
    call GrowEFG(efg, P=mknode("+"))
    call BuildEFG(efgT=&P, sdg.true, cc)
    call BuildEFG(efgF=&P, sdg.false, cc)
    call GrowEFG(efgT, Q=mknode("end"))
    call GrowEFG(efgF, Q)
    efg = P.end = Q
  else if sdg is a relevant WHILE node then
    call GrowEFG(efg, R=mknode("*"))
    P = Q = nil
    If sdg.test is relevant then
      call BuildEFG(efg, sdg.test, cc)
      call GrowEFG(efg, P=mknode("-"))
      R.test = P
    endif
    If sdg.body is relevant then
      call BuildEFG(efg, sdg. body, cc)
      call GrowEFG(efg, Q=mknode("end"))
      R.end = Q ; set P.end to Q if P is not nil
    endif
  else U sdg is a relevant or event CALL node then
    call GrowEFG(efg, P=mknode(PrintFunction(sdg, "in")))
    copy sdg's relevant or event marking into P
    call BuildEFG(efg, FindEntry(sdg), sdg.calling_context)
    If efg is pointing at P then   /* empty function body */
      P.printlabel = PrintFunction(sdg, "call")
    else
      call GrowEFG(efg, Q=mknode(PrintFunction(sdg, "out")))
      P.end = Q
    endif
  else if sdg is an event then
    call GrowEFG(efg, mknode(sdg.printlabel))
  endif
end
```

Figure 18. The EFG construction algorithm.

43. The View Generation and Refinement Component

In this section, we complete the discussion of the construction framework by describing the use of EFG for view generation and refinement. In section 4.1, we motivated the EFG by listing the additional requirements that generating and refining the program views entail. Here, we shall see how the EFG meets these requirements, and allows the regular expression and abstract control flow views to be generated and refined in a simple and straightforward way.

43.1. Generating and Refining the Regular Expression view

As Figures 2 and 14 clearly show, the regular expression view *is* a direct representation of the EFG; its generation therefore requires only the direct translation of the EFG into the box notations shown in Figure 3. Similarly, regular sub-expressions are simply paths from the EFG's *start* vertex to the selected EFG vertex.

The view translation process is simple: walk the EFG and for each vertex, determine its presentation and its location in the view. *Event* vertices are presented according to their *printlabels*, and the presentation for the ''+'' and ''*'' vertices are shown in Figure 3. The main task in the translation is the positioning of the *event* vertices' presentation so that the composition sequence is reflected. We represent the position of an event vertex's presentation area by the quadruple

(wnum, xpos, ypos, len)

which indicates the area's starting position and length in the logical window labeled wnum. Any presentation tool would need to deal with the problem of presenting large views on small screens; our solution is to employ a hierarchy of logical windows to present sub-expressions that are too deeply nested. That is, expressions beyond a certain nesting level (level 3 or 4 for a 80-column display window) in a window are presented as elided expressions, whose presentation is continued in a separate window, labeled wnum. The positions of event vertices are then computed with respect to a logical window. This computation is implementation-dependent; an illustration of the process for Figure 2 is given in Figure 19.

Composition Sequence	if $E_1.pos$ = (w, x, y, l), *then*
E_1, E_2	E_2.pos = (w, x, y+1, l)
$E_1, (E_2 + E_3)$	E_2.pos = (w, x, y+2, (l-1)/2)
	E_3.pos = (w, x+1+l/2, y+2, (l-1)/2)
$E_1, (E_2)^*$	E_2.pos = (w, x+3, y+4, l-2)

Figure 19. Computing an event's logical position in a window.

43.2. Generating and Refining the Abstract Control Flow (source) view

Figures 4 and 14 show the close correspondence between the EFG and the abstract control flow (source) view. The view generation process is, again, straightforward. Walk the EFG path corresponding to the selected regular sub-expression; the source representation of' the *event*

vertices on this path are presented, the ''+'' and ''*'' vertices help determine their syntactic structure, and the *func.in* and *func.out* vertices determine the inlining of calls. (Recall that each EFG vertex points at its corresponding vertex in the SDG; hence the source representation can be directly retrieved.) Hierarchical source elision is easily implemented with the aid of pairing edges.

5. Constructing the CPF, PWOGV and MECS Abstractors

The basic contribution of the construction framework is the ease with which abstractors for a wide class of programming plans can be constructed. In this section, we describe the `IdentifyEvent` routines that are needed to build the abstractors for the CPF, PWOGV and MECS programming plans from the construction framework.

5.1. Calling Pattern of Functions

This programming plan is the simplest of the three plans, yet it can be surprisingly versatile. We have shown that the pattern of file I/O operations is an instance of the CPF plan. By varying the starting function f_{start}, and the set of functions fset whose calling pattern from f_{start} is desired, other plans instances can be obtained. For example, by setting f_{start} to be some some function `A`, and fset to a set of utility functions, we get a plan instance describing how the set of utility functions are used by `A`.

The event identification requires only knowledge of the function call relations. We assume that each function is assigned a unique integer code, and that each function f s entry vertex in the SDG is annotated with a bit vector calls that represents the set of functions that are transitively reachable from f via function calls. Identifying events for this programming plan is then effected by the routines given in Figure 20. Notice the obligations listed in section 4.2.1 are fulfilled here: (1) No actions are needed for non-call-site statements, since by default they are non-,events, (2) only call sites are involved, (2) printlabels are specified and (3) the function set *fset* is used as a ''constant'' calling context.

```
procedure FunctionCallingPatternAbstractor()
declare
  sdg: a SDG vertex
  fset: calling context is a set of functions
  start: the EFG start vertex
  end: a pointer to EFG vertices; will point at end vertex on return
begin
  sdg = the function entry vertex of f_start in SDG
  fset = the set of functions whose calling pattern is desired
  start = mknode("start")
  call BuildEFG(end=&start, sdg, fset)
end

procedure IdentifyEvent(f, fset)
Input
  f: the function entry vertex of the function whose events are to be identified
  fset: the set of functions whose calling pattern is desired
declare
  c: a call site vertex in SDG
begin
  for each call site node c in f's PDG do
    if callee function at c ∈ fset then
      /* reached a function in fset */
      mark c as an event
      set c.printlabel to the callee function's name
      c.callingcontext = fset
    else if FindEntry(c).calls ∩ fset ≠ O then
      /* called a function that can transitively reach one in fset */
      mark c as relevant
      c.callingcontext = fset
    endif
  endfor
end
```

Figure 20. The routines for constructing the CPF abstractor from the framework.

53. Program-Wide Occurrence of Global Variables

The `IdentifyRoutine` for the PWOGV plan requires information about direct uses and definitions of variables at assignment statements, and potential side effects at call sites. The former is local information that can be obtained from traditional data flow analysis. We denote the set of variables directly used and defined at a statement s, by DUSE(s) and DDEF(s) respectively. Potential side effects at a call sites has been described in section 3; the corresponding side effects at a call site c are denoted USE(c) and MOD(c) respectively. Figure 21 describes the event identification process using these sets. Again, all obligations are fulfilled: (1) Only direct definitions or uses of *gvar* can be events, (2) call sites are relevant if gvar may be modified or used during its invocation, (3) event printlabels describe the function and the alias, if applicable, and (4) the alias set of *gvar* fR is used as the calling context.

Computing new aliases at call sites for this plan is somewhat more interesting.. Here, we follow Cooper's alias analysis work [Coop85]. Alias analysis is performed in two stages: alias introduction and alias propagation. The first detects the start of an alias to gvar in a called function due to gvar being passed directly as an actual parameter, whereas the second tracks an alias chain by tracing the passing of gvar's aliases to a called function. Figure 22 describes this computation.

```
procedure GlobalVariableOccurrenceAbstractor()
declare
  sdg: a SDG vertex
  aliar: calling context is a set of variable aliases
  start: the EFG start vertex
  end: a pointer to EFG vertices; will point at end vertex on return
begin
  sdg = entry vertex of program in SDG
  aliar = O /* initial calling context is the empty alias set */
  start = mknode("start")
  call BuildEFG(end=&start, sdg, alias)
end

procedure IdentifyEvent(f, alias)
input
  f: the function entry node of the function whose events are to be identified
  alias: a set of f's formal parameters {fp_1, ..., fp_k} that are aliased to gvar
declare
  v, w: SDG vertices
  WorkList: a set of SDG vertices
  fp: a formal parameter off (aliased to gvar, fp ∈ alias)
begin
  /* Uses a worklist to walk all vertices in f */
  WorkList = {f}
  while WorkList ≠ O do
    extract vertex v from WorkList
    if v is a CALL vertex then
      if either MOD(v) or USE(v) contains gvar or anyfp ∈ aliar then
          mark v as relevant
          /* compute new aliases as calling context for callee function */
          v.calling_context = ComputeNewAliases(v, alias)
      endlf
    else
      if DDEF(v) contains gvar or any fp ∈ alias then
          mark v as an event
          set v.printlabel to "gvar@f=" or "fp@fl/gvar="
      else if DUSE(v) contains gvar or any fp ∈ alias then
          mark v as an event
          set v.printlabel to "=gvar@f" or "=fp@fl/gvar"
      endlf
      add to WorkList all vertices w such that v →_cd w
   endlf
  endwhile
end
```

Figure 21. The routines for constructing the **PWOGV abstractor**

```
function ComputeNewAliases(call_site, alias)
Input
  call-site: a call-site vertex in SDG
  alias: a set of caller's formal parameters {fp_1, ..., fp_k} that are possible aliases to gvar
returns
  newalias: a set of callee's formal parameters {fp'_1, ..., fp'_t} that are possible aliases to gvar at call-site
declare
  ap: an actual parameter of call_site
begin
  newalias = O
  /* Alias Introduction (passing gvar as actual parameters) */
  if gvar is passed as an actual parameter at call_site then
    for each actual parameter ap in which gvar occurs do
      add to newalias the callee's formal parameter bound to ap
    endfor
  endif
  /* Alias Propagation (passing gvar's aliases as actual parameters) */
  for each fp_i ∈ alias passed as an actual parameter at call-site do
    for each actual parameter ap in which fp_i occurs do
      add to newalias the callee's formal parameter bound to ap
    endfor
  endfor
  return newalias
end
```

Figure 22. Compute callee's aliases to *gvar* at *call-site,* from the caller's aliases, via introduction or propagation.

53. Modification Effects at Call Sites

The key effort in specifying `IdentifyRoutine` for the MECS plan for a call site c_{orig} is in the tracking of the modification effects at c_{orig} (i.e. the actual parameters and global variables that may be modified at c_{orig}), across functions, to their definition points in other functions. To do this, the calling context to be carried into a callee function is structured as a set of bindings between the modification effects at c_{orig}, to the formal parameters of the callee. On entry into the callee, for each binding, the reaching definitions of the formal parameter are traced. If assignments are found, they are labeled with the modification effects; if side effects at call sites are involved, a new binding is created and added to the call site's calling context. The process is then repeated at call sites. Note that aliases to a modification effect me, are now indicated by the presence of multiple bindings to *me,* in the calling context. Figure 23 implements these ideas.

```
procedure ModificationEffectsAbstractor()
declare
  sdg: a SDG vertex
  me_fp_bindings: the calling context is a set of binding tuples (me_j, fp_{i_j}), where me, is a modification effect at the
      call site, and fp_{i_j} is a formal parameter of functions called from the call site
  start: the EFG start vertex
  end: a pointer to EFG vertices; will point at end vertex on return
begin
  sdg = the function entry vertex of the function called at call site c_orig
  me_fp_bindings = {(me_1, fp_{i_1}), ..., (me_k, fp_{i_k})}, representing the set of bindings of actual parameters and global
              variables me_i at c_orig, to the corresponding formal parameters fp_{i_j} of the function called at c_orig
  start = mknode("start")
  call BuildEFG(end=&start, sdg, me_fp_bindings)
end

procedure IdentifyEvent(f, me_fp_bindings)
input
  f: the function entry node of the function whose events are to be identified
  me_fp_bindings: set of bindings of modification effects at c_orig to formal parameters off
declare
  c: a call site vertex inf
  def: a definition vertex inf
begin
  fix each binding (me_j, fp_{i_j}) ∈ me&–bindings do
    fix each def such that def →_dd fp_{i_j} do
      if def ∈ MOD(c) then
        mark call site c as relevant      I* def is a side effect at call site c *I
        add to c.calling_context the binding (me_j, fp'_{i_j}), where fp'_{i_j} is the callee's formal parameter bound to fp_{i_j} at c
      eke
        mark def as an event              I* def is an assignment statement */
        set def.printlabel to "fp_{i_j}@fl/me_j="
      endif
    eadfor
  eadfor
end
```

Figure 23. Routines for constructing the MECS abstractor.

6. Handling Recursion

The basis of our abstractors is the use of the regular expression as an abstraction of plans; its use is also the source of a limitation, namely, that regular expression cannot describe recursive constructs [AhSe86]. Hence, recursive calls to function cannot be directly abstracted by `BuildEFG`. Our approach to handling recursion is similar to that used in the UNIX cflow call graph generator: Recursive calls are detected during the EFG construction and appear in the regular expression view as special annotations, with pointers to the first occurrence of that function. Figure 24 describes how `BuildEFG`'s actions on first entry into a function can be modified to handle recursion.

```
if sdg is an Entry node then
  if sdg is not marked "entered" then
    mark sdg "entered" and set sdg→pos to efg /* note the current EFG position */
    call IdentifyEvent(sdg, cc)
    (add mark relevant nodes)
    call BuildEFG(efg , sdg→seq, cc)
    unmark sdg
  else                          /* recursion detected */
    call GrowEFG(efg , mknode(PrintRecursion(sdg, sdg→pos)))
  endif
```

Figure 24. Modifying the `BuildEFG` to detect recursion.

7. Related Work

In this section, we review current software engineering tool support for program understanding, and compare our work to other programming-plan-based program understanding approaches.

Many code understanding tools today can generally be categorized as ***program database browsers*** or ***multiple-view abstraction systems,*** or may contain features found in both. Program database browsers provide direct access to a database of program facts through some query language, and many are based on commercial database management systems. Examples include the **Interscope [TeMa81], OMEGA [Lint84], EDSA** [VaCu89], and the CIA **[ChNi90].** Multiple-view abstraction systems, on the other hand, generally compute some abstraction of the program facts, and typically present them as multiple graphical abstracted views. Examples include **PECAN** [Reis84], **MAGPIE** [DeMe84], **program slices [Weis84]** and **TuringTool [CoEl90].** While these tools clearly provide much information to programmers, the information described is generally ***about*** program components, or ***about*** relationships between components, but they do not facilitate the "weaving" together of diverse interrelated components and program relationships necessary for understanding programming plans, especially delocalized ones. To take the PWOGV plan of a global variable as an example, a program database browser can describe its data type declaration, name its abstract data type operations, even list where it is used and defined, and an abstraction system can present graphical views of the program's call graphs, control flow graphs and data dependence graphs; yet, even with all these tool support, the programmer effort required to to reconstruct the PWOGV plan is at best tedious.

Previous efforts at exploiting programming plans as a direct program understanding aid falls into two categories: the ***knowledge-based automutic recognition*** approach, or the ***hypertext-based explicit documentation*** approach. The first approach uses knowledge-based techniques to match the structural and dependence properties in the program against a knowledge base of programming plans, to identify the plans used in the program. The recognized plans can then be explained, expert-system-style, to programmers [HaJi88, RiWi90]. In practice, problems abound. Among the most serious are the effort to adequately populate the knowlegde base [HaJi88] and the computational expense of the search required [RiWi90]. The second approach

advocates the explicit documentation of such plans as the developers* responsibility, and seeks to exploit hypertext technology for ease of information retrieval [DeSc86, SoPi88]. However, besides the obvious burden of documentating plans on developers, our inability to validate the documentation's fidelity makes this approach very error-prone.

We feel that these two approaches represent the extremes. Our approach, which can be termed the ***dependence-based automatic abstraction of designer-specified plans,*** represents a balance between the two, and combines the advantages of both, while avoiding their pitfalls. Like the recognition approach, but unlike the documentation approach, our plan abstractions are automatic and less error-prone, and places no burden on the end-users or developers. Like the documentation approach, but unlike the recognition approach, our method is practical and usable now, without relying on expected breakthroughs.

8. Conclusion

The ability to understand programs at the level of programming plans represents a significant improvement over current program understanding tools, which can only support understanding at the level of component attributes or program relationships. We have presented a methodology which represents a good first step towards making this possible. In addition, the solutions to the technical problem of constructing an abstractor construction framework has been presented, including the algorithms and key data structures used. Our next step is to implement the framework, and construct the various plan abstractors from it to investigate issues related to the practical use of the programming plans in program understanding.

References

[Adel81] B. Adelson, "Problem solving and the development of abstract categories in programming languages", ***Memory*** & ***Cognition,*** Vol. 9, No. 4, 1981, pp. 422-433.

[AhSe86] A. V. Aho, R. Sethi and J. D. Ullman, ***Compilers: Principles, Techniques and Tools,*** Addison-Wesley, 1986.

[BaGr90] R A. Ballance, S. L. Graham and M. L. Van De Vanter, "The Pan Language-Based Editing System For Integrated Development Environments", Proc. of the 4th SIGSOFT Symp. on Software Development Environments, December 1990, pp.77-93.

[ChNi90] Y. F. Chen, M. Nishimoto and C. V. Ramamoorthy, "The C Information Abstraction System", ***IEEE Trans. Softw. Eng.,*** Vol. SE-16, No. 3, March 1990, pp. 325-334.

[CoEl90] J. R. Cordy, N. L. Eliot and M. G. Robertson, "TuringTool: A User Interface to Aid in Software Maintenance", ***IEEE Trans. Softw. Eng.,*** Vol. SE-16, No. 3, March 1990, pp. 294-301.

[CoKe88] K. D. Cooper and K. Kennedy, "Interprocedural Side-Effect Analysis in Linear Time", Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, June 1988, pp 57-66.

[Coop85] K. D. Cooper, "Analyzing Aliases of Reference Formal Parameters", Proc. ACM SIGPLAN Conf. on Principles of programming Language, 1985, pp 281-290.

[DeMe84] N. M. Delisle, D. E. Menicosy and M. D. Schwartz, "Viewing a Programming Environment as a Single Tool", Proc. ACM SIGSOFT/SIGPLAN Symposium on Practical Softw. Eng. Development Environments, April 1984, pp 49-56.

[DeSc86] N. Delisle, M. Schwartz, "Neptune: A Hypertext System for CAD Applications", Proc. ACM SIGMOD, 1986, pp. 132-143.

[HaJi88] M. T. Harandi and J. Q. Ning, "PAT: A Knowledge-Based Program Analysis Tool," Proc. IEEE Conf. on Softw. Maintenance, 1988, pp. 312-318.

[HoRe88] S. Horwitz, T. Reps and D. Binkley, "Interprocedural Slicing Using Dependence Graphs", Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, June 1988, pp 35-46.

[LeSo86] S. Letovsky and E. Soloway, "Delocalized Plans and Program Comprehension", *IEEE Software*, Vol. 3, May 1986, pp. 4149.

[LiPi86] D. C. Littman, J. Pinto, S. Letovsky and E. Soloway, "Mental Models and Software Maintenance". Empirical Studies of Programmers: Proc. of the Fist Workshop, E. Soloway and S. Iyengar (eds), Norwood, N.J., Ablex Publishing Corp. 1986, pp. 80-98.

[Lint84] M. A. Linton, "Implementing Relational Views of Programs". Proc. ACM SIGSOFTISIGPLAN Symposium on Practical Softw. Eng. Development Environments. April 1984. pp 132-140.

[NaSh73] I. Nassi and B. Shneiderman, "FlowChart Techniques for Structured Programming". *SIGPLAN Notices*. Vol. 8, No. 8, August 1973, pp. 12-26.

[SoEh84] E. Soloway and K. Ehrlich,, "Empirical Studies of Programming Knowledge". *IEEE Trans. Softw. Eng.* , Vol. SE-10.1984, pp. 595-607.

[SoPi88] E. Soloway, J. Pinto, S. Letovsky, D. Littman and R. Lampert, "Designing Documentstion To Compensate for Delocalized PLans," *Comm.* ACM., Vol. 31, No. 11, November 1988. pp. 1259-1267.

[Reis84] S. P. Reiss, "Graphical Program Development with PECAN Program Development System". Proc. ACM SIGSOFT/SIGPLAN Symposium on Practical Softw. Eng. Development Environments. April 1984, pp 3041.

[RiMa92] D. J. Richardson, T. O. O'Malley, C. T. Moore and S. L. Aha, "Developing and Integrating ProDA in the Arcadia Environment". Proc. of the 5th SIGSOFT Symp. on Software Development Environments. December 1992. pp. 109-119.

[RiWi90] C. Rich and L. M. Wills. "Recognizing a Program's Design: A Graph-Parsing Approach". *IEEE Software*, January 1990, pp. 82-89.

[TeMa81] W. Teitelman and L. Masinter. "The Interlisp Programming Environment", IEEE Computer, April 1981, pp 25-33.

[VaCu89] L. I. Vanek and M. N. Culp, "Static Analysis of Program Source using EDSA," Proc. IEEE Conf. on Softw. Maintenance, October 1989, pp. 192-199.

[Weis84] M. Weiser, "Program Slicing, " *IEEE Trans. Softw. Eng.* , Vol. SE-10, No. 4, July 1984, pp. 352-357.

Appendix: The Program Dependence Representations

This appendix contains a full definition of the program dependence graph and system dependence graphs introduced in section 3.

Many versions of dependence graphs already exist, and each is an explicit representation of selected data and control dependences within a function. Our representation scheme corresponds most closely to that used in [HoRe88]. Local data and control dependences in functions are represented in a ***program dependence graph*** (PDG) for each function; these PDGs are then connected by call invocation and parameter-linkage edges to form the ***system dependence graph*** (SDG) representation of the entire program.

To simplify the presentation here, we will first define the program dependence graph for a single-function program (i.e. a program without function calls), then introduce function calls and parameter passing to extend this definition to a system dependence graph representation of a multi-function program.

8.1. Program Dependence Graph for Single-Function Program

A program dependence graph for a a single-function program P, denoted by PDG_P, is a directed graph $(V(PDG_P), E(PDG_P))$. The vertex set $V(PDG_P)$ is composed of the subset $V_c(PDG_P)$ that represents statements in P that can affect its control flow, and the subset $V_v(PDG_P)$ that represents statements that can define or use values in P. Specifically, $V_c(PDG_P)$ consists of the ***seq***, if, ***whib*** vertices to represent the usual control flow operations, and the program ***entry*** vertex, while $V_v(PDG_P)$ consists of vertices to represent assignment statements and predicates in control statements in P.

The edge set $E(PDG_P)$ is similarly composed of two subsets: the subset $E_c(PDG_P)$ that represents the control dependences in P, and the subset $E_v(PDG_P)$ that represents the loop-independent and loop-carried data dependences in P. Control dependences are represented as edges from control-flow-affecting vertices to vertices in general, that is, $E_c(PDG_P) \subset V_c(PDG_P) \times V(PDG_P)$. A control dependence edge from vertex v, to vertex v is denoted by v, $\rightarrow_{cd}$ v, and may be labeled **true** or **false**, or it may be unlabeled. A labeled edge (from if and ***while*** vertices) indicates that v will be executed only if v, is executed ***and*** the predicate associated with it evaluates to a result matching the label; an unlabeled edge (from ***seq*** or program ***entry*** vertices) indicates that v will always be executed if v_c is executed.

Data dependences are represented as edges from assignment vertices to value-affecting vertices, that is, $E_v(PDG_P) \subset V_v(PDG_P) \times V_v(PDG_P)$. A data dependence edge from an assignment vertex $v_=$ to vertex v, indicates that a variable is defined at $v_=$ that may be used at v, and an execution path exists from $v_=$ to v that does not redefine the variable. If this execution path consists of backedges, then the data dependences edge is loop-carried and is denoted by $v_= \rightarrow_{dd_{lc}}$ v; otherwise it is loop-independent and is denoted by $v_= \rightarrow_{dd_{li}}$ v.

The construction of program dependence graphs in general has been described elsewhere. [FeOte87] gives a comprehensive and formal description of the PDG construction algorithm, with

special emphasis on the construction of control dependences for languages with unstructured constructs. [OtEl92] details the implementation of, and experience with, a Fortran-to-PDG translator. For our simplified language model, our control dependences correspond to the syntax structure of the program, and hence can be constructed in the same manner as the program's abstract syntax tree [AhSe86]. Data dependences are computed from the traditional reaching definitions analysis, which for our language model, can also be performed using a syntax-directed approach

83. System Dependence Graph of Multi-Function Program

The ***system dependence graph*** (SDG) representation of a multi-function program is essentially a "supergraph" consisting of the PDGs of each function in the program, extended to represent function calls and parameter passing. In this section, we detail these extensions to the PDG, and describe how these extended PDGs are connected together in a way that reflects the function call structure and parameter linkages.

8.2.1. Extending the PDG to represent function calls and parameter passing

The main extension to a function's PDG needed to accommodate function calls and parameter passing is the representation of "hidden" interprocedural side-effects not shown explicitly in a function. These effects includes the initial-definitions and final-uses of formal parameters, as well as the possible-uses and possible-definitions of actual parameters at call sites. Identification of such sideeffects is made difficult if parameters are passed by reference and global variables are permitted, since these sideeffects may now involve global variables not even present in the function. In such cases, to discover what variables may be used or modified as a result of invoking a function, ***interprocedural flow-insensitive side-effect analysis*** [CoKe88] of the program is required. This analysis annotates every function f in the program with two variable sets $GMOD(f)$ and $GUSE(f)$, each containing all the formal parameters and global variables that may be modified and used, respectively, as a result of f's invocation.[3] Once the *GMOD* and *GUSE* sets of functions in the program are known, identification of side-effects in a function f is easy. $GMOD(f)$ and $GUSE(f)$ directly determine the set of initial-definitions and final-uses present in f, and the set of possible-definitions and possible-uses at a call site c that calls function g can be determined from $GMOD(g)$, $GUSE(g)$ and the parameter bindings at c. Following the terminology of [CoKe88], we will denote by $MOD(c)$ and $USE(c)$ respectively, the set of possible-definitions and possible-uses of actual parameters and global variables at a call site c.

More formally, the presence of function calls and parameter passing in a function f requires its PDG representation to be extended by adding these new vertices:

- **The control-flow-affecting vertex subset $V_c(PDG_f)$ now includes the function** entry vertex **and function call vertices; the first to represent transfers of control into** f **(and back out to the caller), and the second to represent transfers of control out of** f **(and back from the callee).**

[3] $GMOD(f)$ and $GUSE(f)$ are defined in [CoKe88] to contain all variables that may be modified and used, and may include local variables of f. For our purposes, local variables are irrelevant, and we will restrict $GMOD(f)$ and $GUSE(f)$ to variables that may be visible outside f, namely, the formal parameters and global variables.

- Six new types of vertices are added to $V_v(PDG_f)$:

  (1) A ***side-effect-definition*** vertex to represent an initial-definition at function entry or a possible-definition at a call site. That is, each variable in $GUSECf)$, and each variable in $MOD(c)$ of a call site c in f, will be represented by a ***side-effect-definition*** vertex labeled "$v=$", where v is the variable represented.

  (2) A ***side-effect-use*** vertex to represent a final-use at function exit or a possible-use at a call site. That is, ***each*** variable in $GMOD(f)$, and each variable in $USE(c)$ of a call site c in f will be represented by a ***side-effect-use*** vertex label "$=v$", where v is the variable represented.

  (3) For convenience, two distinguished vertices ***GMOD*** and ***GUSE*** are used to represent *the* $GMOD(f)$ and $GUSE(f)$ sets in f. These vertices are control dependent only on f's function ***entry*** vertex. All ***side-effect-definition*** vertices representing variables in $GMOD(f)$ are made control dependent on the ***GMOD*** vertex, and all ***side-effect-use*** vertices representing variables in $GUSE(f)$ are made control dependent on the ***GUSE*** vertex.

  (4) Similarly, each call site c in f will have a ***USE*** and ***MOD*** vertex each to represent the $USE(c)$ and $MOD(c)$ sets. They are control dependent only on c's function ***call*** vertex. Again, all *the* ***side-effectdewtion*** vertices representing variables in $MOD(c)$ are made control dependent on *the* ***MOD*** vertex, and all the ***side-effect-use*** vertices representing variables in $USE(c)$ are made control dependent on the ***USE*** vertex.

8.2.2. Connecting the extended PDGs to form the SDG

The second and final step in building a system dependence graph is to connect the extended PDGs together in a fashion that reflects their function call structure and parameter linkages. To this end, we introduce two types of interprocedural edges, ***call-invocation*** edges and ***parameter-linkage*** edges, and use them to connect PDG_{caller}'s to PDG_{callee}'s into a "supergraph", as follows:

(1) A ***call-invocation*** edge is used to link every ***call*** vertex of a PDG_{caller} to the function ***entry*** vertex of the PDG_{callee}.

(2) A ***parameter-linkage*** edge is used to wnnect every ***side-effect-use*** vertex control dependent on a USE vertex in PDG_{caller}, to the corresponding ***side-effect-definition*** vertex control dependent on the GUSE vertex of PDG_{callee}. These edges therefore represent the passing of values from the caller into the callee, just before invoking the call.

(3) Another ***parameter-linkage*** edge is used to wnnect every ***side-effect-use*** vertex control dependent on a GMOD vertex in PDG_{caller}, to the corresponding ***side-effect-definition*** vertex control dependent on the MOD vertex in PDG_{callee}. These edges therefore represent the passing of values from the callee back out from the ***callee,*** just before retuming from the call.

Figure 1 of this appendix shows the full source representation of the pedagogical example given in sction 2.1. The source statements of assignments, predicate expressions and parameters are included to provide examples of the data dependences and parameter linkages. Figure 2 shows the full SDG for this program. The three PDGs corresponding to each function f are enclosed in the dotted boxes. Vertices with boldfaced labels are vertices from the vertex subset $V_c(PDG_f)$; the rest form the $V_v(PDG_f)$ vertex subset. Single dashed arrows represent control dependences in the edge subset $E_c(PDG_f)$, while loop-independent and loop-carried data

dependences in $E_v(PDG_f)$ are shown as solid and double arcs respectively. *Call invocation* and *parameter-linkage* edges are shown as double dashed arrows, and double solid arrows respectively.

```
program ()                    FUNC1 (int sum, cnt)        FUNC2 (int s, n)
{int sum, cnt;                {int n;                     {
 cnt = read();  /*stat1*/      n = cnt + 1;   /*stat5*/     s = s + n ;  /*stat7*/
 sum = 0;   /*stat2*/         while (n > 0)( /*pred2*/     write(n);    /*stat8*/
 if (cnt > 0){  /*pred1*/      n = n - 1;    /*stat6*/      if (n <= 0) /*pred4*/
  FUNC1(sum, cnt);             if (even(n)) /*pred3*/       write(s); /*stat9*/
  write(sum); /*stat3*/         FUNC2 (sum,n);            }
 }                             }
 else                         }
  write(cnt); /*stat4*/
}
```

Figure 1. A "full source" version of the pedagpgical example of section 2.1.

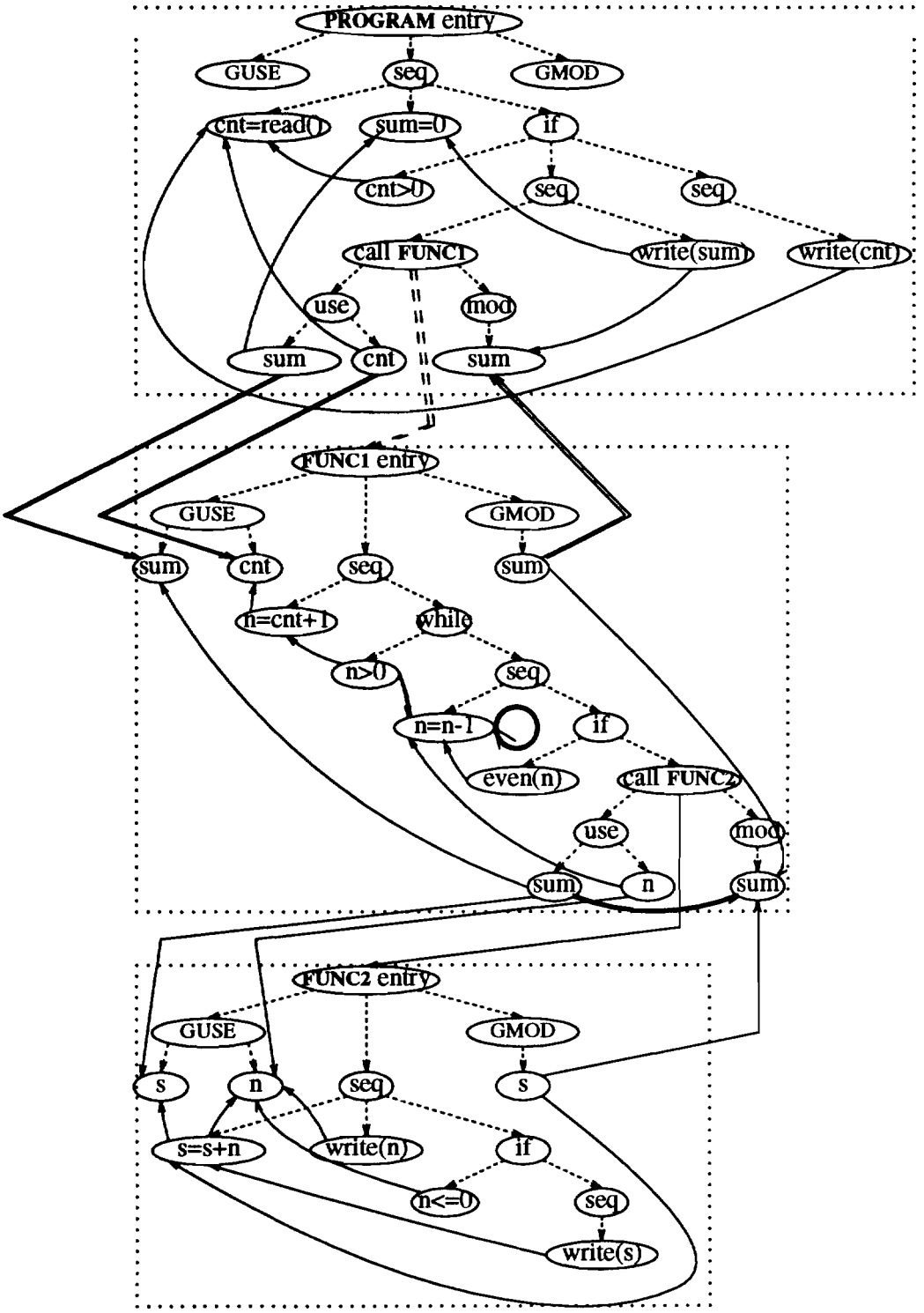

Figure 2. The SDG for the program of Figure 1.