A Toolset for Program Understanding

Panos E. Livadas
Computer and Information Sciences
University of Florida
Gainesville, FL 32611

Scott D. Alden
Computer and Information Sciences
University of Florida
Gainesville, FL 32611

Abstract

Program understanding is associated with the hierarchy of abstractions and interpretations that are deduced from the code [16]. Therefore, the understanding process parallels that of the bottom-up programming process in that maintainers begin by associating small groupings of individual instructions with higher-level interpretations. The understanding process is repeated until the desired level of understanding is attained.

Program understanding in this context requires the identification and study of the numerous complex interrelationships that are induced by the *data flow, calling*, and *functional* dependencies that exist in the software. Therefore, an environment is needed in order aid the programmer in understanding software. The *internal program representation* (IPR) plays a critical role in the nature of that environment.

In an earlier paper, we discussed both an internal program representation and an environment that conforms to the requirements stated above.[11] The internal program representation, the system dependence graph (SDG), is a directed labeled multigraph that captures all control and data dependences, as well as the calling context of procedures; it is based on the one proposed in [8]. The toolset is referred to as *Ghinsu* and it supports a number of tasks over a program written in a subset of ANSI C such as slicing, dicing, and ripple analysis.

In this paper we will present some background on the problems associated with program understanding and show how the *Ghinsu* toolset can aid the programmer in understanding software.

1 Introduction

Software maintenance is an expensive, demanding, and ongoing process. Boehm [3] has estimated that one US Air Force system cost $30 per instruction to develop and $4,000 per instruction to maintain over its lifetime. This case may be exceptional, but the maintenance costs for a large embedded system seem to be an average of two to four times the development costs. It is generally recognized that the primary reason that software maintenance is so costly is that each modification requires first and foremost that the software be *understood.* A program is said to be understood if an overall interpretation of the program is achieved. Most of the proposed models fall into one of the two categories: code-driven (bottom-up) or problem-driven (top-down) [4, 5, 15].

We hypothesize that both strategies are employed by the programmers engaged in this activity. We also support the notion that different maintenance tasks require different *kinds* of program understanding, and therefore different processes are required. As an example, consider the code segment below.

```
( 1).       sum = 0;
( 2).       read(next);
( 3).       while (next>=0) do
( 4).           begin
( 5).               sum = sum + next;
( 6).               read(next);
( 7).       /*              do_something(next); */
( 8).           end;
( 9).           write(sum);
(10).           write(next);
```

We can identify several different kinds of program understanding such as the following:

1. Understanding that the definitions of **sum** that can directly affect the use of **sum** at statement (5) are at statements (1) and (5).
2. Understanding that the definitions that directly affect statement (5) are at statements (1), (2), (5), and (6).
3. Understanding that the value of **sum** at statement (9) depends on the statements (1), (2), (3), (5), and (6).
4. Understanding that the segment adds up a number of input values until it reads a negative value, after which it prints the result and the last value read.
5. Understanding that the author assumes that the source of input values is non-empty.
6. Understanding that the segment determines the sum of all scores in the recent exam of the class CS201.

This example illustrates the two principal domains of program understanding: the *programming* and the *application* domain. The first four kinds of understanding belong to the programming domain, and represent an extrapolation of the code's intent in terms of standard programming interpretations and problem solving techniques. The fifth kind belongs to the application domain, and differs from the rest because it represents an abstraction of the code's intent in terms of a specific application. This domain lies outside of the domain of program interpretations, and requires program documentation for understanding.

Program understanding is associated with the hierarchy of abstractions and interpretations that are deduced from the code [16]. Therefore, the understanding process parallels that of the bottom-up programming process in that maintainers begin by associating small groupings of individual instructions with higher-level interpretations. The understanding process is repeated until the desired level of understanding is attained.

Program understanding in this context requires the identification and study of the numerous complex inter-relationships that are induced by the *data flow, calling*, and *functional* dependencies that exist in the software. Program segments are not just as simple as the example above may erroneously indicate. The example contains only localized interactions. As Letovsky and Soloway [9] have established, programmers have difficulty understanding code that has non-local interactions. For example, if the call to procedure **do_something** is uncommented, it is not clear which of the definitions of **sum** can reach the use of **sum** at statement number (5). The answer to this question depends on whether or not the variable **sum** is defined (as a global variable) in the body of the procedure **do_something**, or by some other procedure that **do_something** invokes (directly or indirectly) before it returns.

Given the complexity of the task, is not surprising therefore that programmers spend approximately 60% of the maintenance time ”looking at” code [19]. Therefore one can conclude that maintenance quality and productivity can be improved by supplying the maintainer with a set of proper tools that he/she may employ for understanding the target software.

On the other hand, a number of organizations have found that simply purchasing new tools does not automatically increase productivity [1]. What is needed is the creation of a process for each type of understanding that uses a set of tools designed within the framework of this process. These tools should allow the maintainer to ask questions about a program and be provided with *precise* answers.

In order to study tool-assisted program understanding, we must provide an effective environment for understanding programs. Such an environment should be integrated with the existing software maintenance tools and should provide additional facilities to support other software engineering activities. We have already developed much of this environment, but some research issues remain. For example, the answers that the understanding tool provides should be presented to the programmer in a way that best improves the maintainer's understanding of the program relationships. This task is not trivial because of the large amount of dependencies in a software.

Furthermore, the maintenance tools should have a fast interactive response time. Otherwise, the maintainers will be discouraged from using them.

Realizing the need for such an environment, two and half years ago we embarked upon the task of developing such an environment and tools and turning the theoretical concepts into practical realities, with the support of the Software Engineering Research Center[1]. We have made considerable progress towards these goals.

The key element of our system is its internal program representation (IPR), the *System Dependence Graph*, or SDG. The SDG is a parse tree representation of the program. The nodes represent program constructs, in and out parameters, call-sites, etc. The edges represent various kind of dependencies (such as data flow, control flow, and declaration) among the nodes to which they are adjacent. The main benefit of this structure is that it represents a vast amount of information that could be shared by numerous software engineering tasks. These applications typically use the same kind of information, but use different representations. Our approach eliminates the redundancies. Since all algorithms use the SDG as the underlying structure, they are source language independent.

The environment, referred to as *Ghinsu*, supports a number of tasks such as program slicing, dicing, ripple analysis, dependency analysis, DU-chain, UD-chain, and reaching definitions calculation as well as a host of browsing activities.

The remainder of this paper is organized as follows. The next section presents background on slicing, dicing, and ripple analysis. We then briefly describe the internal program representation and how it is derived. Finally we present a tour of the *Ghinsu* toolset and show some of its major functions and tools.

2 Background

From our perspective, the most important concept is that of a static slice, since it is used to build the SDG. In addition, a slicing tool can provide useful information for the software maintainer.

Slicing provides a way to decompose a large program into smaller, independent components. Let P be a program, p be a statement in P, and V be a subset of the variables of P. Weiser defines as a *static slice* of P relative to the *slicing criterion* $< p, V >$ to be the set of all statements and predicates of P that *might* affect the values of variables in V at the statement p. Weiser reports experimental results that experienced programmers use slicing when debugging [17]. Weiser found that programmers remembered the slice relevant to the bug as having been used or probably having been used in almost half the cases examined. When debugging, programmers view programs in ways that need not conform to the program's textual or modular structures. In particular, the statements in a slice may be scattered throughout the code of the larger program. Yet, experienced programmers routinely abstract these slices from a program. Weiser concluded that since programmers remembered the relevant slices from the program they had debugged, they were probably mentally constructing and using these slices while debugging. Presumably each programmer had independently developed the slicing method. If novice programmers were taught the concept of slicing, they could avoid this reinvention and learn debugging techniques faster.

Since debugging is a process in which programmers try to better understand code to find and eliminate bugs, and since programmers find slices when debugging, it is logical that a tool that automatically creates program slices would be useful not only in debugging but also in code understanding [17].

Suppose that during testing, we find that the value of a certain variable, v, is incorrectly computed at statement n. By obtaining a slice of v at n, we may extract a significantly smaller piece of code than the entire program in which to locate the bug. If the value of another variable, w, is computed correctly at statement n, then we may employ a method that was suggested in [13] and is referred to as *dicing* (computing the intersection of two slices). The bug is likely to be associated with one or more statements in a set referred to as the *Fault Prone Statement Set (FPSS)*, which is the set of statements associated with the slice on v minus those associated with the slice on w. The FPSS is obtained by generating the complement of the slice on w relative to the slice on v.

If a large program computes the value of a variable x and the code associated with this function is needed in another application, then one could slice on this variable and use the extracted program in the latter application. Therefore, program slicing aids in code reuse.

[1]Funded by the National Science Foundation, the Center's 15 industrial affiliates, and the Florida High Technology and Industry Council.

The number of slices, their spatial arrangement, etc., may hold significant information about the structuring of a program [17]. Hence, an assortment of *program metrics* can be computed and their actual significance investigated. Useful metrics include coverage, overlap, clustering, and tightness.

Ripple analysis identifies the statements that will be affected when a change is to be made at a given statement (i.e, ripple analysis is "forward" slicing). A program maintainer can examine the ripple of a statement to help determine the possible effects of a proposed modification.

Structured walkthroughs and code inspection activities would be easier to perform by calculating interprocedural reaching definitions (the set of statements s' which reach a statement s) , *DU-chains* (a chain that links a use to all definitions that may reach it), and *UD-chains* (a chain that links a definition to all of its possible uses).

Furthermore, run-time support can be provided through *automatic data generation* (by using the calculated UD and DU-chain information). *Dynamic slicing*, and other pertinent tools that can be built by using the SDG to instrument the generated code. Most of our tools can be run on incomplete programs provided that they are compilable. Hence, these tools can be used even at the development stage.

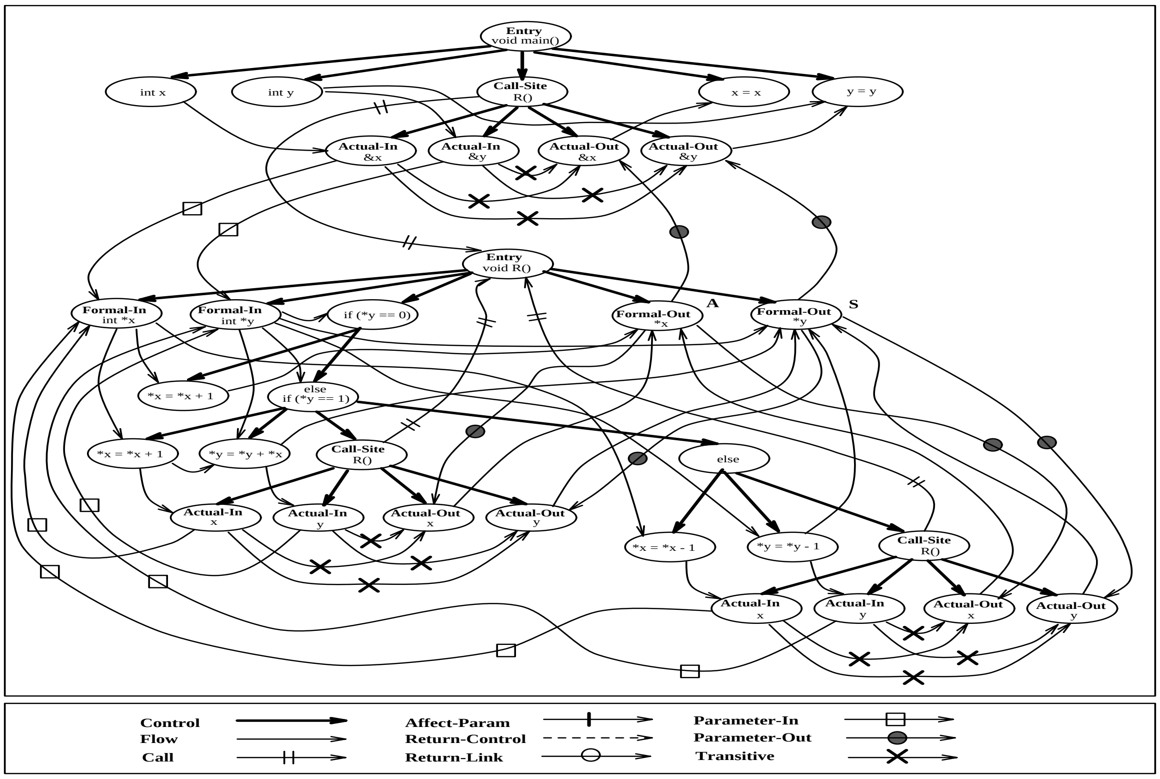


Figure 1: The SDG corresponding to the code in Figure 4.

We have also developed an *object finder* that uses information from the SDG to group together related types, data, and routines [10, 12]. We note that this tool can help objectify code and capture the objects that the original designer had in mind.

3 The Internal Program Representation

Weiser's slicers [18] were based on a flow-graph representation of programs. Ottenstein et al. [14], show that an *intraprocedural* slice could be found in linear time by traversing a suitable graph representation of the program that they referred to as the program dependence graph (PDG). Horwitz et al. [8] introduce algorithms to construct *interprocedural* slices by extending the program dependence graph to a supergraph of the PDG, which is referred to as the system dependence graph (SDG). This extension captures the calling context of the procedures which was lacking in the method proposed by Weiser.

This new approach not only permits more precise slices than [18], it also permits slicing when the program contains calls to *unknown* procedures (procedures whose bodies are not available), provided that the *transitive dependencies* (discussed later) are known. As was pointed out in [14], the internal program representation (IPR) chosen plays a critical role in the software development environment. An example of a SDG is shown in Figure 1

We have developed a prototype that accepts programs written in ANSI C or Pascal and generates a *parse tree based* SDG. We have implemented tools such as a slicer, dicer, ripple analyzer, dependency analyzer, DU-chains, UD-chains, a reaching definitions calculator (even if these definitions or chains span procedure boundaries), and a browser that utilize this SDG.

The grammar proposed in [7] consists of a single (main) program and supports scalar variables, assignment statements, conditional statements, and while loops, but does not support variable declarations. The language consists of a collection of procedures whose parameters are passed by value-result, and which end with `return` statements. These `return` statements can not be arbitrarily located in the procedure, nor can they actually return values to their calling procedure(s).

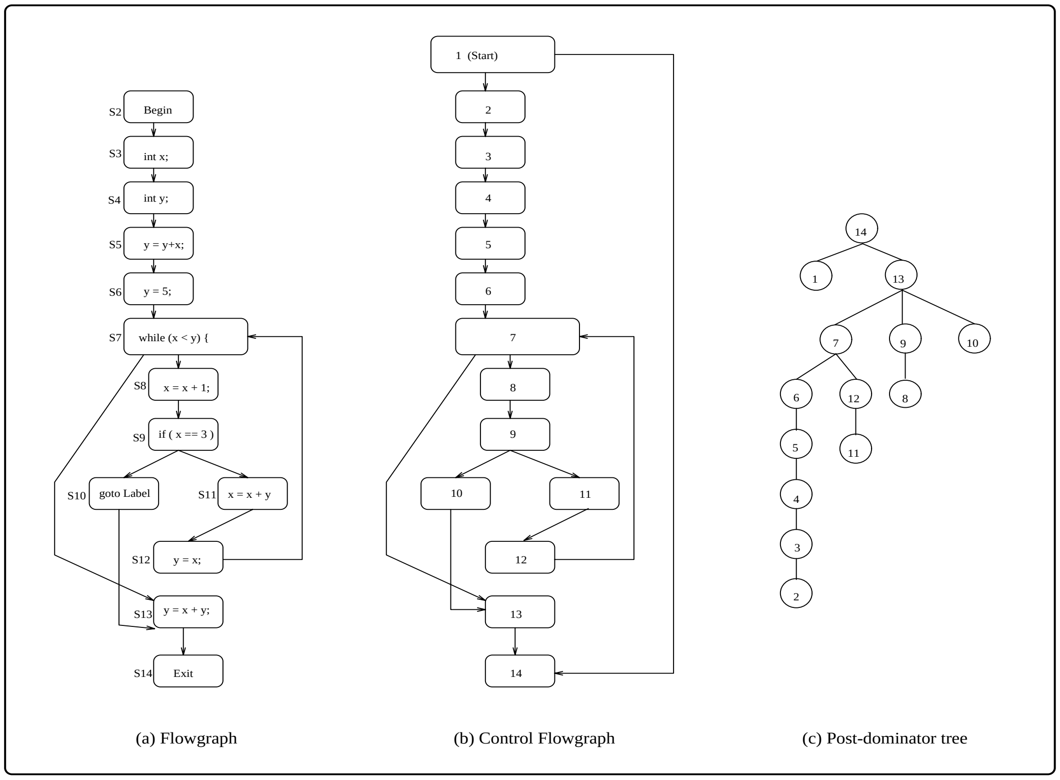
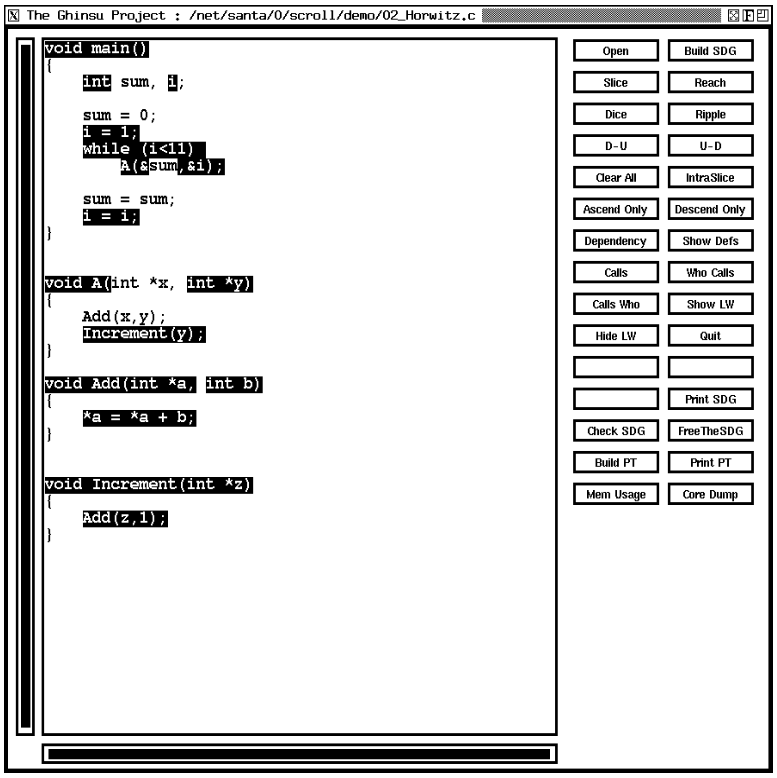


Figure 2:

We extend this grammar and consequently modify the SDG as follows: First, variable declarations are supported. Second, we distinguish between pass-by-value and pass-by-reference parameters. The same notation is employed as in C, in order to determine the type of parameter passing. However, pointer operations are restricted to those that constitute pass-by-reference parameters; i.e., if `x` and `y` are pointer variables, we permit assignments of these variables such as `*x = 4` and `*y = *x` (where `*` denotes a de-referencing of the contents of the variables). Third, any number of `return` statements are permitted to appear anywhere in a procedure. These `return` statements can contain expressions that may include variables and are modeled after the `return` statements in C. Fourth, we distinguish among functions that return values as opposed to those that do not. Fifth, all C constructs are handled except `long jumps`. Finally, we use a *parse tree* as the basis of our SDG. This allows slicing to be more precise than if the "resolution" of the SDG was only at the statement level.

Even though the SDG and the slicing algorithm are based on the work proposed in [8], our methods are considerable extensions of previous works. First, our grammar is a superset of the grammar targeted in [8]. Second, our method of building the SDG differs in many respects. Our method eliminates the need to compute the GMOD and GREF sets of each procedure in the system and to construct a linkage grammar and its corresponding subordinate characteristic graphs of the linkage grammar's nonterminals. Third, we use a *parse tree* as the basis of our SDG. This allows slicing to be more precise than if resolution of the SDG was only at the statement level. The improved precision occurs because the algorithm for slicing [6] requires the traversal of certain edges backwards. Hence, when a statement such as `x = x + y +foo(&a)` is encountered during the computation of the slice of `a`, then the union of the slices associated with `x` and `y` will be included in the slice of `a`. It is also clear that variables `x` and `y` do not affect the value of `a`. By employing a parse-tree-based SDG, we are able to avoid this shortcoming of statement-based SDGs and therefore arrive at more precise slices and therefore more precise data dependence analysis (since the latter depends on the former).



```
void main()
{
    int sum, i;

    sum = 0;
    i = 1;
    while (i<11)
        A(&sum,&i);

    sum = sum;
    i = i;
}


void A(int *x, int *y)
{
    Add(x,y);
    Increment(y);
}

void Add(int *a, int b)
{
    *a = *a + b;
}


void Increment(int *z)
{
    Add(z,1);
}
```

Figure 3: The slice relative to the statement `i=i`

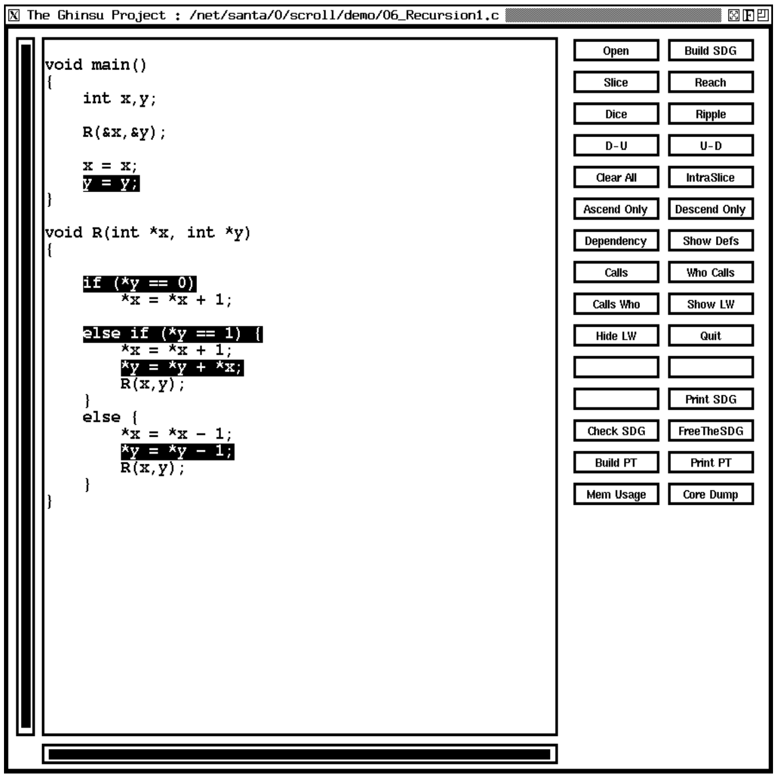
3.1 Control and Data Dependence

Control Dependence A flowgraph (program graph) is a directed graph with an initial node from which all other nodes can be reached. Nodes correspond to basic blocks and edges represent transfers of control between basic blocks. Figure 2(a) shows an example flowgraph whose initial node has been marked by "Begin." Dependences among blocks arise as the result of either control or data dependences.

A node y is said to be control dependent on x if there exists a directed path from x to y such that every node z on the path (not including x or y) is post-dominated by y, and y does not post-dominate x.

In order to calculate control dependences, a control flowgraph of a program is needed. The control flowgraph's post-dominators are then calculated which is equivalent to calculating the dominators of the reverse control flowgraph (all edges of control flowgraph are reversed). We then use the algorithm discussed in [6] to compute the control dependences.

Figure 2(b) and 2(c) show an example of the control flowgraph and its corresponding post-dominator tree.



```
void main()
{
    int x,y;

    R(&x,&y);

    x = x;
    y = y;
}

void R(int *x, int *y)
{

    if (*y == 0)
        *x = *x + 1;

    else if (*y == 1) {
        *x = *x + 1;
        *y = *y + *x;
        R(x,y);
    }
    else {
        *x = *x - 1;
        *y = *y - 1;
        R(x,y);
    }
}
```

Figure 4: The DU-chain at statement `*y=*y+*x`.

Data Dependence In order to compute the data dependences of a program, we must first calculate the reaching definitions for the entire program. We define the *GEN* and *KILL* sets [2] for each block on flowgraph. We then use the iterative algorithm presented in [2] to calculate the reaching definitions. A node x is data flow dependent from a node y if node y defines a variable that is used in em x.

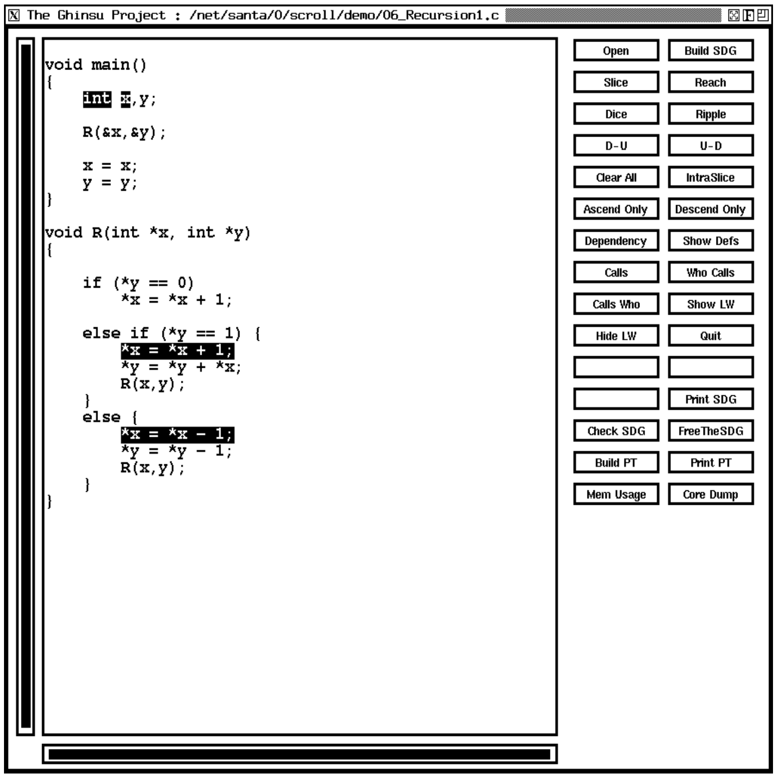
We now consider the case of routine invocation. When a call-site is encountered, the flowgraph is annotated by introducing actual_in and actual_out nodes ([11]). The actual_out nodes are considered as unknown (U-nodes)[11] until the time the called procedure is solved. The assignment of the actual_out nodes depends on the corresponding formal_out nodes. Specifically, if a formal_out node is an A-node, we consider its corresponding actual_out node to be a definition. If it is an I-node, the actual_out node is considered to be a definition, but its *KILL* set is defined to be empty. Finally if the formal_out node is an IN-node, then both its *GEN* and *KILL* sets are empty by definition.

When we encounter a call-site, we suspend solution of the current procedure and descend into the called procedure and calculation of the reaching definitions is begun there. This procedure is repeated until one procedure calls no other procedure. At this point, all of the data flow dependences of the last encountered procedure can be calculated since its *GEN* and *KILL* sets are known.

4 A Tour of Ghinsu

In this section we briefly present the Ghinsu environment and the tools that we have implemented. Ghinsu accepts a source program written in a subset of either ANSI C or Pascal as input and produces the SDG as described earlier. This SDG can subsequently be used by any of the available tools.

Figure 3 presents a simple graphical user interface that we developed using X-Window library routines that facilitates user interaction with the system. A brief description of the major components of Ghinsu as well as the tools discussed earlier follows. It should be noted that except for YACC all components were built "from scratch".



```
void main()
{
    int x,y;

    R(&x,&y);

    x = x;
    y = y;
}

void R(int *x, int *y)
{

    if (*y == 0)
        *x = *x + 1;

    else if (*y == 1) {
        *x = *x + 1;
        *y = *y + *x;
        R(x,y);
    }
    else {
        *x = *x - 1;
        *y = *y - 1;
        R(x,y);
    }
}
```

Figure 5: The UD-chain at statement `*x=*x-1`.

YACC: The parser generator YACC is used in the Ghinsu project to generate a parser that when fed an ANSI C or Pascal source program produces a parse tree as output. This step produces the nodes and control flow edges and is the skeletal structure on which the rest of the system dependence graph is built. Each terminal node is annotated with its corresponding location in the source (a line and column number). This allows us to achieve a mapping from the source file to/from the SDG.

Dependency Generator: The dependency generator takes the parse tree (generated by the YACC) as its input and produces the parse tree based system dependence graph. Figure 1 illustrates the statement[2] based SDG produced by the dependency generator for the program shown in Figure 4.

TOOLS: The tools reside here and will be discussed shortly.

Graphical Interface: After the Ghinsu tool has been invoked, the user is presented with a window (not shown) where the files of the given subdirectory are displayed. The subdirectory can be changed by changing the `Path` field in this screen. The file that contains the desired program can be selected by clicking on its filename and subsequently can be opened via `open` button. Before any of the tools are invoked the user must request that the SDG corresponding to the file opened should be built; this is accomplished via the `build SDG` button. At this point, the user must position the cursor on the target statement (and variable) that he/she wishes to inspect; then he/she should invoke the appropriate module (slicer, dicer, ripple analyzer, dependence analyzer, DU-chain, etc.)[3].

The mapping between the source code, the graphical display, and the SDG is straightforward. When the user selects some variable on a statement to have some action performed on it (such as a slice), the line and column is

[2]We illustrate the statement based SDG as opposed to the parse tree based for the sake of simplicity.

[3]The `Clear All` button is employed to clear all highlighted text; the `Quit` button is used to exit from Ghinsu. Buttons that are not discussed are used for the tool's development process. Additionally, the `object finder` button is not shown.

determined. The SDG is then searched for a match based on the line and column. If a match is found, the variable is highlighted; and, its corresponding node is "remembered". If the user subsequently chooses an action, the node remembered is used as the target node. The results of the action are reflected on the display by traversing the SDG and highlighting the source corresponding to the nodes that are marked (e.g., in the slice).

Slicer: This module calculates slices on a system dependence graph. The screen dump shown in Figure 3 illustrates the slice of the program relative to the statement `i=i` by highlighting the statements that belong to that slice. In this context, the maintainer may use the *intraprocedural slice* button whenever he/she wishes to limit his/her view to the scope of one function. Furthermore, two more buttons related to slicing are provided. The *ascend only* tool allows the maintainer to limit the slice to only the function selected and the functions that call the selected function. This operation corresponds to slicing phase one only. Correspondingly, the *descend only* tool allows the maintainer to limit the slice to only the function selected and the functions called by the selected function. This operation corresponds to slicing phase two.

DU-chain: Two more screen dumps are displayed. Both illustrate the *precision* as well as the identification of the *position* of the uses and definition of variables at a given statement even if procedure boundaries are crossed and even if recursive procedures are present. In Figure 4 the statement `*y=*y+*x` has been selected and the DU-chain has been requested, i.e., the determination of all uses for this definition of `*y`. Notice that `*y` is used in the predicate `(*y == 0)`; if this statement evaluates to false, it will be used in the predicate `(*y==1)`. Furthermore, if the latter predicate evaluates to false, then the variable will be used at the statement `*y=*y-1`. Finally, since there is an execution path that passes through the statement `*y=*y+*x` and statement `y=y`, the latter statement is captured since `y` is used there.

Similarly, Figure 5 illustrates the definitions of the variable `x` that can reach its use in statement `*x=*x-1`. In that case what reaches this statement is either the declaration (ud-anomaly can therefore be detected) or the statement containing the definition `*x=*x+1` depending of course on the data.

Calls: This button invokes a tool that displays the calling sequence.In addition, the user could query the system via either the *who calls* or *calls who* buttons. Specifically, the maintainer selects (via the cursor) a function such as `sample`. In the former case, all functions that call the function `sample` will be identified whereas the functions that are invoked by the function `sample` will be identified.

Dependency: This button invokes the data flow dependence analyzer. It is assumed that already a statement has been selected as we described earlier.The output indicates the line number, variable name, type of variable and the function in which each variable that may affect the value of the selected variable is visible.

Finally, the *Show definitions* selection causes all statements to be identified at which a maintainer-specified variable has been defined.

5 Acknowledgements

We wish to acknowledge Steve Croll for the implementation of the *Ghinsu* Toolset.

References

[1] SERC Industrial Affiliates. Personal communication, 1989-1992.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, Mass.

[3] B.W. Boehm. The High Cost of Software, Practical Strategies for Developing Large Software Systems, E. Horowitz (ed.). *Addison-Wesley* Reading, Mass.

[4] V. Basili and H. Mills. Understanding and documenting programs. *IEEE Transactions on Software Engineering*, SE-8(3):270–283, 1982.

[5] R. Brooks. Towards a theory of the comprehension of computer programs. *Int'l J. Man-Machine Studies*, 18:543–554, 1983.

[6] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3):319–349, 1987.

[7] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. In *Proc. 15th ACM Symposium of Programming Languages*, 1988.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, January 1990.

[9] S. Letovsky and E. Soloway. Strategies for documenting delocalized plans. In *Proc. Conf. on Software Maintenance*, pages 144–151, 1985.

[10] P.E. Livadas and P. Roy. Program dependence analysis. In *IEEE Conf. on Software Maintenance*, 1992.

[11] P.E. Livadas and S. Croll. System Dependence Graphs Based on Parse Trees and their Use in Software Maintenance. In *Journal of Information Sciences*, (to appear).

[12] P.E. Livadas and T. Johnson. A new approach to finding objects in programs. Technical Report cis.ufl.edu:cis/tech-reports/tr92/tr92-037.ps.Z, U. Florida Dept. of CIS, 1992.

[13] J.R. Lyle and M. Weiser. Automatic program bug location by program slicing. In *Proc. 2nd International Conference on Computers and Applications*, 1987.

[14] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. ACM SIGPLAN Notices 19,5, 1984.

[15] R.E. Seviora. Knowledge-based program debugging systems. *IEEE Software*, 4(3):20–32, 1987.

[16] J. Wedo. Structured program analysis applied to software maintenance. In *Proc. Conf. on Software Maintenance*, pages 28–34, 1985.

[17] M. Weiser. Programmers use slices when debugging. *CACM*, 1982.

[18] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 1984.

[19] N. Wilde. SDTC Lecture Series 1: Software Development Environments. CENET, October 1992.