IJREAT International Journal of Research in Engineering & Advanced Technology, Volume 2, Issue 3, June-July, 2014
ISSN: 2320 – 8791 (Impact Factor: 1.479)
www.ijreat.org

Development Of Software Inspection Tool By Performing Slicing

Rakhee Kundu[1], Umesh Kulkarni[2]

[1]Computer Engineering, ARMIET, Affiliated to Mumbai University, India

[2]Computer Engineering, VIT, Affiliated to Mumbai University, India

ABSTRACT : Although software inspection has led to improvements in software quality, many software systems continue to be deployed with unacceptable numbers of errors, even when software inspection is part of the development process. The difficulty of manually verifying that the software under inspection conforms to the rules is partly to blame. We describe the design and development of a tool designed to help alleviate this problem. The tool provides mechanisms for inspection of software by exposing the results of sophisticated whole-program static analysis to the inspector. The tool computes many static-semantic representations of the program, forward and backward slicing and dependence factors. Whole-program pointer analysis is used to make sure that the representation is precise with respect to aliases induced by pointer usage. Views on the dependency and related representations are supported. Queries on the dependence graph allow an inspector to answer detailed questions about the semantics of the program. Facilities for openness and extensibility permit the tool to be integrated with many software-development processes. The main challenge of the approach is to provide facilities to navigate and manage the enormous complexity of the dependence graph. Which will test the correctness of the program by identifying some of the rules .Whether particular variable in the program is working or malfunctioning, Checking the malfunctioning by the dependency factors by using backward and forward slicing. This will identify the checkpoints and not to identify the errors and which area a particular checkpoint is getting effected will be reflected.

***Keywords* – Abstract Syntax Tree, Program Dependence Graph (PDG), Predecessor, Slicing, Successor.**

1. Introduction

1.1 Why use testing?

"Testing can consume over 50 percent of software development costs (note that testing costs should not include debugging and rework costs). In one particular case, NASA's Apollo program, 80 percent of the total software development effort was incurred by testing."Some projects cand t afford any failures at all during operation like the Apollo project. It can also be that the customers accept some faults that are fixed later under maintenance because the product will be cheaper. So the time spent during testing much depends on what reliability level that are asked for. One advantage with testing can be that it is closer to the way the end-user will use the system. They will fell that the product has a higher quality because the defects is outside the normal execution. The program will become more reliable by finding the most common failures.

Inspection will more look for correctness there a more common executed fault and a more rarely fault is equally easy to find. I have found in the project that I have been involved in that the test phase often has less priority than other phases. If the project is getting late is it likely that the time for testing will be cut down. This can especially be a problem if the test phase is located at the end of the project and not during the entire project. Inspection can be effective when same method can be used on a lot of different documents and testing is effective when it comes to rerunning the same test.

A good tool for automated testing can take some time to develop but can be executed many times. This can save a lot of time because many systems today are released over and over again. One problem with automated tools is that we need to write additional code and we will not know if the defect is in the systems code or in the test code. There exist areas where testing is the only option and where you canđ t find the defects with inspection. Example is tests that test if the system has the right quality attributes with performance and stress testing. But it is also when the developer doesn't have access to the code such as with third party software components and different APIđ s. Different environments outside the program must also be tested, for example running Java applications in different operative systems or applets in different browsers. But there are also things that you can't find with testing such as lack of traceability, design faults etc. It can also be easy to miss faults in code that is not normally executed like exceptions handing.

1.2. Nothing is perfect

Inspections focuses on finding faults, whereas testing mainly focuses on finding failures (which are the result of one or many faults). They are more compliments to each other than competing methods because they are used to find different faults and in different areas.

Developers will probably need to use both inspection and testing to achieve a product with high quality and still be within budget. " Software inspections can identify and eliminate approximately 80 percent of all software defects during development. When inspections are combined with normal testing practices, defects in fielded software can be reduced by a factor of 10." However, researches have shown that the order in which the inspection and the testing are performed will affect the number of defects found. The best way, according to these researches, are to make inspection first and after that the testing. Thus, What we have found when reading about the two methods is that both is good and must be used to achieve a product that has a high quality and satisfies the end users. They are used for different reasons and in different phases during the project. Each technique has its advantage and way of approaching the search for defects. Their respective strengths help finding different kinds of defects.

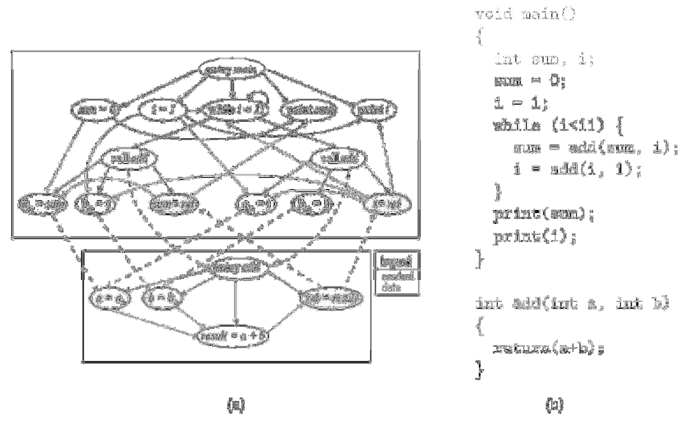
Inspections are better for finding errors in design, requirements documents, source code etc. Testing is the only way of finding operational defects, and to make sure that non-functional requirements are working as they are supposed to.

Tests cannot find errors in requirements documents or in the source code. It depends on previous experience and knowledge about the problem domain among the team members which method that is used. The people performing an inspection may not have the necessary knowledge about the product domain or they may be overloaded with information in the initial stage of the inspection, etc then defects can easily be missed. Testers don't have the same problem because they have test cases to follow.

2. Dependence Graphs

Dependence graphs have applications in a wide range of activities, including parallelization , optimization , reverse engineering, program testing , and software assurance . Fig. 1 shows the dependence-graph representation for a simple program with two procedures. This section briefly describes dependence graphs and how they are built.

A Program Dependence Graph (PDG) is a directed graph for a single procedure of a program. The vertices of the graph represent constructs such as assignment statements, call sites, parameter ,and condition branches.



```
void main()
{
  int sum, i;
  sum = 0;
  i = 1;
  while (i<11) {
    sum = add(sum, i);
    i = add(i, 1);
  }
  print(sum);
  print(i);
}

int add(int a, int b)
{
  return(a+b);
}
```

Figure 2.1 Program Dependance Graph

An edge between the vertices indicates either a data dependence or a control dependence. The data-dependence edges indicate possible ways in which data values can be transmitted. For example, in Fig. 1, there is a data dependence edge between the vertex for i=1 and the vertex for while (i < 11), which indicates that a value for i may flow between those two vertices.

A control-dependence edge between a source vertex and a destination vertex indicates that the result of executing the source vertex controls whether or not the destination vertex is reached. For example, in Fig. 1, there is a control dependence edge between the vertex for while (i<11) and the vertices for the two call sites on the function add. A System Dependence Graph (SDG) is a directed graph consisting of interconnected PDGs , one per procedure in the program. Inter procedural control-dependence edges connect procedure call sites to the entry points of called procedures. Inter procedural data-dependence edges represent the flow of data between actual parameters and formal parameters (and return values). Nonlocal variables, such as global, file statics, and variables accessed indirectly through pointers, are handled by modeling the program as if those variables are passed in and out as parameters to the program's procedures. Each nonlocal variable used in a function, either directly or indirectly, is treated as a "hidden" input parameter and, thus, gives rise to additional program points. These serve as the function's local working copy of the nonlocal variable. If the variable is modified in the function, then it has an associated output parameter as well. The process of creating the dependence graph is described in the following sections.

2.1 Front End

For each source file in the system, a language-specific front end is invoked. Its responsibility is to create intermediate files that will be used in subsequent phases:

1. Information from the preprocessor phase, such as the include tree and macro usage, is recorded. Information about the basic structure of the preprocessed source file is recorded. In particular, the line and column numbers of each construct in the source file are recorded.

2. The occurrences and usages of pointer variables are collected.

3. The abstract syntax tree (AST) and symbol table are created. These are then used to create a control-flow graph (CFG).

2.2 Pointer Analysis

The pointer-analysis phase creates the points-to graph for the entire program. A points-to graph is a directed graph with vertices corresponding to variables (and structure fields, arrays, and procedures) and edges indicating the points-to relation between variables . For example, if during program execution x may hold the address of y, then the points-to graph contains an edge from x to y. Heap allocated memory is modeled by introducing one synthetic variable for each occurrence in the program of a construct that allocates memory from the heap. Pointers to heap allocated variables are said to point to these synthetic variables. Constant pointer-valued objects, such as strings, can be modeled either individually, or via a single abstract location which acts as a proxy for them all. The main pointer analysis algorithm implemented is that due to Andersen , with an option to treat structure fields separately. The points-to graph is written out as a database. This database is consulted during phases of the SDG builder.

2.3 The SDG Builder

The SDG builder creates the final dependence graph in several phases. The graph is stored in its entirety in memory that is memory-mapped to a file.

1. A first approximation to the call graph is created by reading the CFGs for all source files, extracting callsite vertices, and connecting them with call edges.

2. The final call graph is created by resolving indirect call sites by consulting the previously created point sto database. An indirect call through a pointer fp is treated as a possible call to all functions in the points-to set of fp.

A depth-first search is then performed on the call graph to partition it into strongly connected components. Several subsequent phases are carried out by traversals over the partitioned call graph, often with an iterative computation carried out on each strongly connected component.

3. The CFGs for each function are read in. The variable usage information computed by the front end is augmented using the information from the points-to database.

4. Information about possible uses and definitions of global variables is computed for each procedure and each call site. The algorithm used is similar to the GUSE/GMOD algorithms of Cooper and Kennedy, except that, to achieve better performance, global variables are partitioned into equivalence classes.

5. An (intra procedural) reaching-definitions algorithm is invoked for each procedure, and the results are used to insert data-dependence edges.

6. The post dominator relations in the CFGs are computed and then used to create the control dependence edges.

7. The CFGs are converted into PDGs, by first converting each CFG vertex to a corresponding PDG vertex. The vertex kind is carried across to the PDG vertex.

The PDGs are then stitched together to create the SDG .

8. Summary edges are computed. A summary edge describes the transitive dependence at calsites between output parameters and input parameters.

Summary edges are an important component of SDGs because they allow inter procedurally precise slicing operations to be performed in time linear in the size of the SDG . The time to compute the summary edges themselves,however, is bounded by O.n3., where n is the maximum number of parameters at any call site. Note that, because nonlocal variables are treated as parameters, n may be as large as the total number of nonlocal variables in the program. This computation is asymptotically the most significant time and space bottle neck in the SDG builder.

9. Finally, the graph is completed by adding reversed edges.

2.4 Managing Complexity

The AST and symbol table are essential for navigating the type structure of the program accurately. The call graph, arguably the most important representation for program understanding, can be viewed directly. The variable use/def information and the results of pointer analysis are very useful for understanding the effects of pointer indirection.

However, the sheer size and complexity of the dependence graph makes it impossible to use in its raw form for software inspection. For example, depending on various build options, even a small 6.5 KLOC program in our benchmark suite can have 22,000 vertices, with over 60,000 edges. One 75 KLOC program has almost 400,000 vertices, with over 2,200,000 edges. Clearly, any tool must offer features to help deal with this complexity. Contents of Variable Usage Sets for Some Example Expressions The vast majority of vertices and edges in the dependence graph have to do with nonlocal variables. As discussed above, nonlocal variables are modeled as hidden parameters to functions. The vertices that correspond to these parameters have kind global-formal and global-actual, each of which represents an equivalence set of nonlocal variables. Unlike vertices that represent expressions or statements, these vertices have no representation in the source code of the program.

Other vertices in the dependence graph are introduced to represent the dependence graph efficiently, or are present to allow query algorithms to be expressed cleanly or to execute quickly. The dependence-graph builder also introduces some synthetic functions. Functions are created to model initialization of variables, and to represent indirect function calls efficiently. Again, these artifacts have no representation in the source code. Below, we discuss techniques for hiding this complexity so that useful information can be extracted.

3. Block Diagram Of The Approach

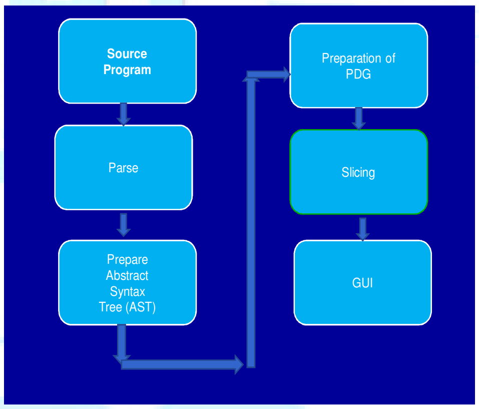


Figure 3.1 Block Diagram Of The Approach

As shown in the figure 3.1 above a source program which is to be inspected is given as an input using GUI, the program is parsed and abstract syntax tree is constructed. The PDG is generated to understand the main flow of the program and then by slicing the program using forward, backward, predessor or successor approach the code is segmented for further analysis. Then CFG algorithm is applied on it to obtain dominator tree and post dominator tree. The control dependence graph is constructed to undertstand the dependancing of a particular variable in the entire program and its linkage with other functions and methods. Using this tool it will be easy to find out bugs in the program and their influence on the program control flow will be understood.

4. System Analysis And Design

4.1 Modularization.

The development of a system generally consists of 2 phases:

A. System analysis.

B. System design.

The development can be thought of as a set of activities that analysts, designer and user carry out to develop and implement the software. Here, the activities are closely related and even the order of the steps in these activities is difficult to determine. However for the sake of easier understanding, the entire project can be viewed as a collection of independent modules.

The modules follow chronological sequence as under:

- Preliminary investigation
- Determination of system requirements
- System analysis
- Design of System
- Development of software
- System testing
- Implementation and evaluation

4.2 Nature of Process Involved

The development of the project involved the following processes:

The Processes

Information Gathering

At this stage, all the initial requirements were gathered and these were clarified for understanding.

Analysis

Here, the requirements were analyzed. These were then categorized so that the incomplete areas are exposed. Finally, the requirements were prioritized by the order of their importance.

Proposal and Project Planning

In this stage, the proposal was developed. Then the project plans were drafted to fulfill the project requirements.

Design

In this stage, the functional description of the project is to be given. Then the project is designed accordingly.

Coding

In this stage, the software coding of the project is done based on the earlier processes. Also a documentation of the project is to be given.

Verification (Testing)

This stage can be thought of as a summation of two processes viz. technical testing and system testing.

4.3 Process Flow Diagram

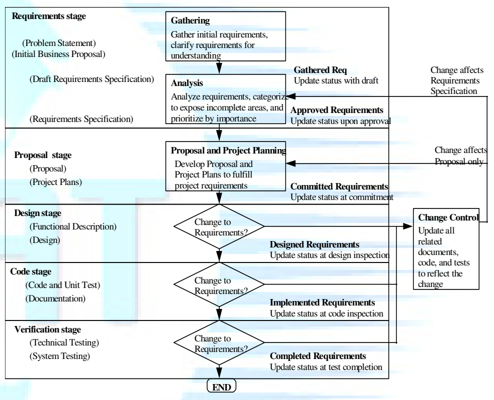

Requirements stage
(Problem Statement)
(Initial Business Proposal)
(Draft Requirements Specification)
(Requirements Specification)
Gathering
Gather initial requirements, clarify requirements for understanding
Analysis
Analyze requirements, categorize to expose incomplete areas, and prioritize by importance
Gathered Req
Update status with draft
Approved Requirements
Update status upon approval
Change affects Requirements Specification
Proposal stage
(Proposal)
(Project Plans)
Proposal and Project Planning
Develop Proposal and Project Plans to fulfill project requirements
Change affects Proposal only
Committed Requirements
Update status at commitment
Design stage
(Functional Description)
(Design)
Change to Requirements?
Designed Requirements
Update status at design inspection
Change Control
Update all related documents, code, and tests to reflect the change
Code stage
(Code and Unit Test)
(Documentation)
Change to Requirements?
Implemented Requirements
Update status at code inspection
Verification stage
(Technical Testing)
(System Testing)
Change to Requirements?
Completed Requirements
Update status at test completion
END

Figure 4.1 Process Flow Diagram

5. Algorithms And Related Theory

5.1 Computation of Basic Blocks

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

We can construct the basic blocks for a program using algorithm GetBasicBlocks, shown in When we analyze a program's intermediate code for the purpose of performing compiler optimizations, a basic block usually consists of a maximal sequence of intermediate code statements. When we analyze source code, a basic block consists of a maximal sequence of source code statements. We often find it more convenient in the latter case, however, to just treat each source code statement as a basic block.

Algorithm GetBasicBlocks

Input. A sequence of program statements.

Output. A list of basic blocks with each statement in exactly one basic block.

2. Construct the basic blocks using the leaders. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

5.2 Computing Control Flow Graph

A *control flow graph* (CFG) is a directed graph in which each node represents a basic block and each edge represents the flow of control between basic blocks. To build a CFG we first build basic blocks, and then we add edges that represent control flow between these basic blocks.

After we have constructed basic blocks, we can construct the CFG for a program using algorithm GetCFG, shown in Figure The algorithm also works for the case where each source statement is treated as a basic block. To illustrate, consider Figure 3, which gives the code for program Sums on the left and the CFG for Sums on the right. Node numbers in the CFG correspond to statement numbers in Sums: in the graph, we treat each statement as a basic block. Each node that represents a transfer of control (i.e., 4 and 7) has two labeled edges emanating from it; all other edges are unlabeled.

In a CFG, if there is an edge from node *Bi* to node *Bj* , we say that *Bj* is a *successor* of *Bi* and that *Bi* is a *predecessor* of *Bj* . In the example, node 4 has successor nodes 5 and 12, and node 4 has predecessor nodes 3 and 11.

Algorithm GetCFG

Input. A list of basic blocks for a program where the first block (*B*1) contains the first program statement.

Output. A list of CFG nodes and edges.

5.3. Computing Dominator Tree

A node D in CFG G *dominates* a node W in G if and only if every directed path from entry to W (not including W) contains D. A *dominator tree* is a tree in which the initial node is the entry node, and each node dominates only its descendants in the tree.

Algorithm ComputeDom

Input. A control flow graph *G* with set of nodes *N* and initial node *n*0.

Output. D(*n*), the set of nodes that dominate *n*, for each node *n* in *G*

5.4 Control Flow Graph (CFG)

A control flow graph describes the sequence in which the different instructions of a program get executed. In other words, a control flow graph describes how the control flows through the program. In order to draw the control flow graph of a program, all the statements of a program must be numbered first. The different numbered statements serve as nodes of the control flow graph . An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node. The CFG for any program can be easily drawn by knowing how to represent the sequence, selection, and iteration type of statements in the CFG. After all, a program is made up from these types of statements.

5.5 Path

A path through a program is a node and edge sequence from the starting node to a terminal node of the control flow graph of a program. There can be more than one terminal node in a program. Writing test cases to cover all the paths of a typical program is impractical. For this reason, the path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths.

5.6 Linearly Independent Path

A linearly independent path is any path through the program that introduces at least one new edge that is not included in any other linearly independent paths. If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent. This is because, any path having a new node automatically implies that it has a new edge. Thus, a path that is subpath of another path is not considered to be a linearly independent path.

5.7 Cyclomatic Complexity

For more complicated programs it is not easy to determine the number of independent paths of the program. McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program. Also, the McCabe's cyclomatic complexity is very simple to compute. Thus, the McCabe's cyclomatic complexity metric provides a practical way of determining the maximum number of linearly independent paths in a program. Though the McCabe's metric does not directly identify the linearly independent paths, but it informs approximately how many paths to look for.

There are three different ways to compute the cyclomatic complexity. The answers computed by the three methods are guaranteed to agree.

Method 1:

Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:

$V(G) = E – N + 2$

where N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

For the CFG of example shown in fig. 10.4, E=7 and N=6. Therefore, the cyclomatic complexity = 7-6+2 = 3.

Method 2:

An alternative way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows:

V(G) = Total number of bounded areas + 1

In the program's control flow graph G, any region enclosed by nodes and edges can be called as a bounded area. This is an easy way to determine the McCabe's cyclomatic complexity. But, what if the graph G is not planar, i.e. however you draw the graph, two or more edges intersect? Actually, it can be shown that structured programs always yield planar graphs. But, presence of GOTO's can easily add intersecting edges. Therefore, for non-structured programs, this way of computing the McCabe's cyclomatic complexity cannot be used.

The number of bounded areas increases with the number of decision paths and loops. Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability. For the CFG example shown in fig. 10.4, from a visual examination of the CFG the number of bounded areas is 2. Therefore the cyclomatic complexity, computing with this method is also 2+1 = 3. This method provides a very easy way of computing the cyclomatic complexity of CFGs, just from a visual examination of the CFG. On the other hand, the other method of computing CFGs is more amenable to automation, i.e. it can be easily coded into a program which can be used to determine the cyclomatic complexities of arbitrary CFGs.

Method 3:

The cyclomatic complexity of a program can also be easily computed by computing the number of decision statements of the program. If N is the number of decision statement of a program, then the McCabe's metric is equal to N+1.

5.8 Program Pependence Graph (PDG)

Different definitions of program dependence representations have been given, depending on the intended application; however, they are all variations on a theme and share the common feature of having explicit representations of both control dependences and data dependences. We define program dependence graphs, which can be used to represent single procedure programs; that is, programs that consist of a single main procedure, with no procedure or function calls. The program dependence graphs (or PDG) for a program P, denoted by GP, is a directed graphs whose vertices are connected by several kinds of edges. The vertices in GP represent the assignment statements and predicates of P. In addition, GP includes a special Entry vertex, and also includes one Initial definition vertex for every variable x that may be used before being defined. (This vertex represents an assignment to the variable from the initial state.) The edges of GP represent control and data dependences.

5.9 Backward Slicing

A backward slice with respect to a set of starting points S answers the question "What points in the program does S depend on?" The control-dependence edges are used to determine how control could have reached S, and the data dependence edges are used to determine how the variables used at S received their values.

5.10 Forward Slicing

A forward slice with respect to a set of starting points S answers the question "What points in the program depend on S?" In this also we make use of control and data dependence edges.

5.11 Predecessors

It is natural for a user attempting to understand a program to ask "How could variable x have gotten its value here?" This query can be posed with respect to the control dependences, the data dependences, or both. A program point's data predecessors are the points where the variables used at that point may have gotten their values.

5.12Successors

It is natural for a user attempting to understand a program to ask "Where is the value generated at this point used next?" This query can be posed with respect to the control dependences, the data dependences, or both. A program point's data successors are the points where the variables that were modified at that point are used.

6. Implementation

The whole system is arranged in the package called **project,** this package contains all the necessary files needed source code and the documentation of the project. The directory contains the two more directories one contains the GUI related code and other contains the back end source code.

7. Conclusion

We have described a tool for inspecting and manipulating the Control flow graph representation of a program for the purposes of program understanding and discussed how it can be used for software inspections. We have described the means by which the system answers queries about the dataflow properties of the program using context-free language graph reachability. We have described using a model checker to answer questions about possible paths through the program. There are two main thrusts in its development. The first is we have improved the scalability of the system. This is achieved partly by using demand-driven techniques to reduce the up-front cost of building the dependence graph. The other thrust is we have extended the domain of applications for the system. We can make any source program efficient by minimizing the dependency graph. The Future scope of this project is one can apply the technology to software assurance, and to program-testing problems.

8. References

[1] L.O. Andersen, "Program Analysis and Specialization for the C Programming Language," PhD thesis, DIKU, Univ. of Copenhagen,May 1994.

[2] T. Ball and S.K. Rajamani, "Bebop: A Symbolic Model Checker for Boolean Programs," Proc. SPIN Workshop, pp. 113-130, 2000.

[3] S. Bates and S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs," Proc. Symp. Princples of Programming Languages, pp. 384-396, 1993.

[4] P. Bishop, R. Bloomfield, S. Guerra, and T. Clement, "Software Criticality Analysis of COTS/SOUP," Proc. Safecomp 2002, Sept. 2002.

[5] M. Burke and R. Cytron, "Interprocedural Dependence Analysis and Parallelization," Proc. SIGPLAN '86 Symp. Compiler Construction,pp. 162-175, 1986.

[6] Bell Canada, http://www.iro.umontreal.ca/labs/gelo/datrix, 2001.

[7] E.M. Clarke, M. Fujita, P.S. Rajan, T. Reps, S. Shankar, and T. Teitelbaum, "Program Slicing of Hardware Description Languages,Proc. Conf. Correct Hardware Design and Verification Methods (CHARME '99), Sept. 1999.

[8] E.M. Clarke, O. Grumberg, and D.A. Peled, Model Checking. MIT Press, 1999.

[9] K.D. Cooper and K. Kennedy, "Interprocedural Side-Effect Analysis in Linear Time," Proc. ACM SIGPLAN 88 Conf. Programming Language Design and Implementation, pp. 57-66, June 1988.

[10] J.R. Cordy, C.D. Halpern, and E. Promislow, "TXL: A Rapid Prototyping System for Programming Language Dialects," Computer Languages, vol. 16, no. 1, pp. 97-107, Jan. 1991.

[11] D.E. Denning and P.J. Denning, "Certification of Programs for Secure Information Flow," Comm. ACM, vol. 20, no. 7, pp. 504-513, July 1977

[12] J. Drake, V. Mashayekhi, J. Riedl, and W. Tsai, "A Distributed Collaborative Software Inspection Tool: Design, Prototype, and Early Trial," Technical Report TR-91-30, Univ. of Minnesota, Aug. 1991.

[13] A. Dunsmore, “Comprehension and Visualisation of Object- Oriented Code for Inspections,” Technical Report EFoCS-33-98, Computer Science Dept., Univ. of Strathclyde, 1998.

[14] D. Engler, B. Chelf, A. Chou, and S. Hallem, “Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions,” Proc. Fourth Symp. Operating Systems Design and Implementation, pp. 1-16, Oct. 2000.

15] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon, “Efficient Algorithms for Model Checking Pushdown Systems,” Computer Aided Verification, pp. 232-247, 2000.