TUFTS-CS Technical Report 2004-9

September 2004

Composed Control Dependence Graph Generator

by

Theodore P Shevlin
Dept. of Computer Science
Tufts University
Medford, Massachusetts 02155

ABSTRACT

Prior to doctoral work by Stafford, control dependence algorithms only worked for uni-procedure analysis and inlined multi-procedure analysis. Inlined multi-procedure analysis fails to address recursion, and in most cases can be too costly to perform. Stafford took a compositional approach to multi-procedure analysis by developing a language-independent, composed control dependence graph for any uni- or multi-procedure software component. This paper details an effort to implement that algorithm in the C++ language using Stanford University Intermediate Format (SUIF) and Machine SUIF (MachSUIF).

Composed Control Dependence Graph Generator

By: Ted Shevlin

Advisor: Judith Stafford

In partial fulfillment of the requirements for a Master of Science in Computer Science

Tufts University Computer Science Department

ACKNOWLEDGMENTS

I would like to thank the following people for their help in bringing this project to fulfillment:

- Judith Stafford, Tufts University. While not only the author of the algorithm described in this paper, she has also helped me to understand it and given me the tools to implement this project. She has provided ample amounts of her time to aiding my development, and she has been very supportive and understanding when I have not been true to my deadlines.
- Glenn Holloway, Harvard University. Glenn has been my single largest source of help in all things related to MachSUIF. While under no obligation to do so, he has examined my code many times and has helped me to debug numerous issues related to MachSUIF. He has been instrumental in my progress, and I may never have achieved what I did with my project were it not for him.
- SERG, Tufts University. The Software Engineering Research Group at Tufts University, composed of (in alphabetical order) Shuanghong He, Srikanth Ravi, Matt Salter, Kevin Simons, Judith Stafford, and myself. They have lent me their time to helping me debug issues related to C, and have offered me advice on coding up some of the more implicit assumptions built into the CCDG algorithm. In particular, I wish to thank Matt Salter, who has sat down with me on a couple of occasions, and Shuanghong He, who has concurrently developed a graphical user interface for my backend, and has been infinitely patient with the changes I've made to my code.

TABLE OF CONTENTS

ACKNOWLEDGMENTS 2
INTRODUCTION 4
INTRODUCTION: OBJECTIVE 5
THE ALGORITHM 6
THE ALGORITHM: DATA STRUCTURES 6
THE ALGORITHM: DATA STRUCTURES: PCFG 6
THE ALGORITHM: DATA STRUCTURES: PFDF 8
THE ALGORITHM: DATA STRUCTURES: PCDG 9
THE ALGORITHM: DATA STRUCTURES: Call Graph 11
THE ALGORITHM: PROCEDURES 12
THE ALGORITHM: PROCEDURES: Construct_PFDF() 12
THE ALGORITHM: PROCEDURES: Construct_PCDG() 12
THE ALGORITHM: PROCEDURES: Construct_CCDG() 13
THE IMPLEMENTATION 16
THE IMPLEMENTATION: API'S USED 16
THE IMPLEMENTATION: INSTALLATION 18
THE IMPLEMENTATION: DATA STRUCTURES 20
THE IMPLEMENTATION: DATA STRUCTURES: Edges 20
THE IMPLEMENTATION: DATA STRUCTURES: Collections 22
THE IMPLEMENTATION: DATA STRUCTURES: Call graph 23
THE IMPLEMENTATION: STEPS TAKEN 24
THE IMPLEMENTATION: STEPS TAKEN: Construct_PFDF() 24
THE IMPLEMENTATION: STEPS TAKEN: Construct_PCDG() 25
THE IMPLEMENTATION: STEPS TAKEN: Construct_CCDG() 27
THE IMPLEMENTATION: THE PROJECT AS A COMPONENT 28
CONCLUSION 31
CONCLUSION: FUTURE WORK 31
CONCLUSION: RELATED WORK 34
REFERENCES 35

INTRODUCTION

Control dependence, in short, describes the ability of a program statement to affect the execution of another program statement. The introduction of procedure calls into control dependence analysis complicates matters significantly. Until recently, dependence analysis on multi-procedure programs could be performed using two different methods: one involving inlining a procedure's control flow graph into the calling procedure, and the other involving joining control flow graphs with edges from the procedures' call sites to their respective target procedures.

Prior to doctoral work performed by Stafford [6], control dependence algorithms worked for uni-procedure analysis. Multi-procedure analysis involving inlining a called procedure's control flow graph does work, as it simply produces a large uni-procedure control flow graph. However, problems occur when the graph grows too large so as to be computationally intractable. Additionally, a control flow graph which inlines the same procedure more than once is going to cause redundant analysis. The inlining method also does not address recursion, which would be impossible to handle when a procedure recurses an unknown number of times (which is generally the case). The other aforementioned method does not take into consideration non-return from the called procedure. Non-return may result from a halt (e.g. an "exit" statement in a C program) embedded in the called procedure, or an infinite loop or infinite recursion.

The method described in Stafford's thesis takes a compositional approach. A compositional approach analyzes each procedure in isolation and then, using the source program's call graph, attempts to combine each procedure's graph in a meaningful manner. In addition to addressing the inefficiency of inlining, Stafford's method also augments the traditional control dependence graph with inherited control dependence arcs, which address the possibility of non-return due to halting or infinite recursion/looping.

INTRODUCTION: OBJECTIVE

The purpose of this project is to provide a proof of concept for Stafford's algorithm. Until this writing, the algorithm has never been committed to code. The eventual hope is that it would provide a tool for developers, for debugging and maintenance purposes; or researchers, for other analyses. Such a tool may require data dependence analysis, which is neither addressed in this project nor in Stafford's algorithm. For right now, to demonstrate that Stafford's algorithm is correct and possible to implement is a good starting point.

This paper describes the algorithm and its implementation, and the problems that occurred during the process and how they were resolved. The project, itself, can be thought of as a software component: it provides a composed control dependence graph (CCDG), and it requires a control flow graph (CFG) and forward dominance information for each source procedure. It used another component from which to derive control flow graph and forward dominance information: MachSUIF, an API designed by Michael D. Smith's research group at Harvard University. MachSUIF, though a good tool in its own right, was not a perfect fit for this project, and to that end, this paper will detail where the project ends and the MachSUIF API begins, should a future project commence to swap MachSUIF out for a more appropriate API.

Before discussing MachSUIF and components, it is important to have a general understanding of the Stafford algorithm.

THE ALGORITHM

The scope of this section is not to describe the algorithm or its background in full. Stafford's thesis provides such an explanation [6]. The purpose of this section is to describe it in enough detail to the reader so that the reader can understand the decisions made when implementing the project.

THE ALGORITHM: DATA STRUCTURES

The idea behind compositional analysis of a source program is to process each individual source procedure separately and then to recombine them afterwards, at what is called "program composition time." Creating the CCDG requires a few data structures beforehand. First is the procedure control flow graph, or PCFG. (In many cases, it is simply referred to as the CFG.)

THE ALGORITHM: DATA STRUCTURES: PCFG

A CFG is a representation of the control flow of a program. A control flow graph works, informally, as follows: if the execution of statement2 has the potential to directly follow the execution of statement1 during the execution of a program, then statement2 is a child of statement1 in the control flow graph. In other words, the CFG maps all possible walks through a program's execution.

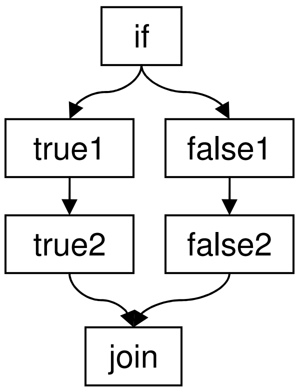

Figure 1. An "if" statement in a program can be modeled in a CFG, as shown above.

Figure 1 shows an “if” statement as modeled in a control flow graph. Nodes “true1” and “true2” are sequential statements that follow in execution if the “if” statement evaluates to true. Likewise, “false1” and “false2” are sequential statements that execute if the “if” statement evaluates to false. The node “join”, while not representing a particular program statement, is necessary to indicate that control flow at that point no longer follows two paths following the “if” statement, but that the paths have merged at that point. A CFG will represent a traditional loop as a set of nodes in sequence, with the last node in the loop extending an arc back to a predecessor node that holds the loop’s conditional statement, or loop header. The aforementioned loop header node will have two arcs, one proceeding down the list of statements to execute following a “true” evaluation, and one jumping to the next statement in sequence after a “false” evaluation.

Stafford augments the traditional PCFG with a new type of arc. In addition to the normal arc, which is the only type of arc shown in Figure 1, she introduces the “interrupted flow arc”, or IFA, which extends from every procedure call to the next node in sequence. (For future reference, note that an “IFA target” is the child vertex in an IFA. I.e., for any IFA (u, v), v is the IFA target. Also, a node representing a procedure call is a “call site”.) The purpose of this arc is to recognize that control flow does not actually move from a called procedure node to the following node, but that it moves from the calling node to an entirely different procedure, and then eventually back to that node. An IFA will later help to recognize the possibility for non-return from a procedure call, which as previously stated, could result from infinite looping, infinite recursion, or an embedded halt.

A final augmentation, which is commonplace for many analyses, is a super initial vertex, V^I, and a super return vertex, V^R. In all cases, except where halts exist, all control flow must initiate from V^I and terminate at V^R. In cases where halts exist, control flow also terminates at the node corresponding to the halt.

THE ALGORITHM: DATA STRUCTURES: PFDF

From the PCFG, one can generate the procedure forward dominance forest, or PFDF. The notion of it being a forest is somewhat of a misnomer, because it may only be a tree, as seen in Figure 1. "Forward dominance", also known as postdominance or inverse dominance, describes the relationship of CFG nodes with regards to execution order in a walk through a procedure or program. Assume that statement1 and statement2 are two statements in a procedure (statement1 and statement2 do not necessarily have to be adjacent to each other). statement2 forward dominances statement1 if every walk from statement1 to the end of that procedure includes statement2.

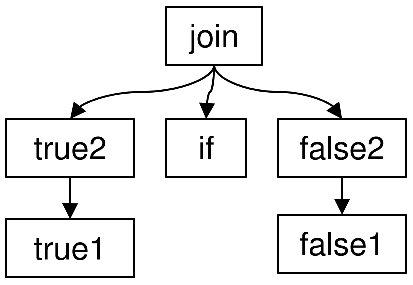

Figure 2. The "if" statement from Figure 1, as a forward dominance forest.

Figure 2, shown above, illustrates the same "if" statement from Figure 1 as a PFDF. As one can see, many of the nodes from the original CFG are in a reverse orientation in the FDF. For instance, "join", which was originally at the bottom of the CFG, is now at the top. That indicates that all walks through the all of the nodes in the original "if" statement must include "join".

It is important to realize that a PFDF is not simply a control flow graph flipped upside-down. Were that to be true, both “true” and both “false” statements would forward dominate the “if” statement, which is clearly not the case. Clearly from the diagram above, “join” is the only forward dominator of “if”, because all walks through “if” must also include “join”. (Some PFDF graphs contain self-arcs, to indicate that all walks from a node include that node. In such graphs, all nodes forward dominate themselves, because all walks through, say, “if”, must include the node for “if”. However, Stafford’s algorithm relies on the notion of “strict dominance”, which does not include those self-arcs.)

Stafford’s PFDF recognizes interrupted flow from procedure calls just as Stafford’s PCFG does. All IFAs in a PCFG have corresponding “reverse” arcs in a PFDF. So, an arc (u, v) in a PCFG will have a special type of arc (v, u) in a PFDF. These reverse-IFAs are known as “pfdom” arcs, or potential forward dominance arcs.

As earlier stated, a PFDF is not always a forest. If Figure 2 represented the entire PFDF for a procedure (and was thus outfitted with vertices V^I and V^R), it would be a tree, not a forest. A PFDF is a forest when a procedure contains one or more halts, because a procedure statement will have two or more possible exit points from that procedure, resulting in two or more roots in the PFDF. (For future reference, the tree/forest type of a particular PFDF will be stated as its “T/F type.”)

THE ALGORITHM: DATA STRUCTURES: PCDG

The last procedure-specific data structure necessary to complete the CCDG algorithm is a PCDG, or “Procedure Control Dependence Graph.” It is interesting to note that a PCDG in Stafford’s algorithm is the equivalent of a CCDG, were the procedure corresponding to the PCDG the only procedure (or "main" procedure) in the program. This is because a PCDG maps control dependence, just as a CCDG does; only it maps it for one procedure.

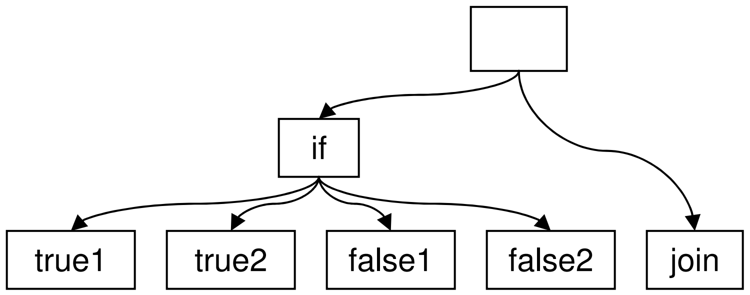

Figure 3. A control dependence graph generated from the CFG in Figure 1, and the FDF in Figure 2.

Figure 3 shows the control dependence graph for the "if" statement used in the previous examples. Control dependence, as previously and informally defined, is the ability of one statement to affect the execution of another statement. As one can see from the above example, the "if" statement has control over the execution of the two branches underneath it in its CFG, however the "join" statement is not control dependent on "if", because it will execute regardless of the evaluation of the condition in the "if" statement. In this particular example, there is no real parent node for "join" and "if" (hence, it is a blank placeholder in Figure 3). However, in a "real" control dependence graph, both the CFG and FDF would be augmented with super entry and exit nodes (V^I and V^R), ensuring that no graph would end up in the situation shown above.

Again, Stafford's algorithm makes modifications to the traditional PCDG. Stafford also adds direct inherited control dependence (dicd) arcs between the parent-child pair of each IFA, and deletes all other arcs leading to IFA target nodes. Additionally, all nodes that forward dominate an IFA target inherit the control dependence of the nearest IFA target that they forward dominate. Stafford indicates this with arcs extending from all IFA targets to their forward-dominating PFDF ancestors. These steps illustrate the uncertainty of return from called procedures. Stafford also inserts potential control dependence arcs from loop headers to their forward-dominating PFDF ancestors to indicate uncertainty of return from loops. (All potential control dependence arcs will be later resolved to "normal" ones or deleted.) Finally, Stafford removes all traditional arcs from the PCDG's super entry vertex, V^I, and replaces them with direct inherited control dependence arcs to indicate that execution of the first procedure statements rely on execution of the procedure, itself.

THE ALGORITHM: DATA STRUCTURES: Call Graph

The final data structure necessary is the program's call graph. Every edge in a call graph represents a call from one procedure to another. For example, an edge (main, A) represents a call originating from main() to the procedure A(). In this implementation, the call graph has a one-to-one correspondence between edges and procedure calls, with information about the call site stored within the edge. So, if main() calls A() twice, there will be two edges, (main, A), with each edge storing information about the original call site, making both edges unique.

The call graph object must also support "contracting", which is a term used to describe the path-based depth-first search algorithm designed by Gabow [2]. The contracting will provide the DFS path used, and thus a post-order traversal path, and will eliminate iteration problems caused by recursion in the source code. (Were one to iterate through a call graph that modeled recursive source code, one would infinitely loop.)

THE ALGORITHM: PROCEDURES

Stafford's algorithm is presented in full in Chapter 4 of her dissertation. This is a brief presentation of that same algorithm.

There are three procedures required to construct the CCDG. Each procedure requires information constructed by the previous procedure to continue. Construct_PFDF(PCFG G), to be executed on each procedure, Construct_PCDG(PCFG G, PFDF F, string CD_type), again to be executed on each procedure, and Construct_CCDG, to be executed once after each PFDF and PCDG has been constructed.

THE ALGORITHM: PROCEDURES: Construct_PFDF()

Construct_PFDF() assumes that the PCFG for the procedure has already been created. The PCFG requires a super entry and super exit vertex and a set of interrupted flow arcs for the set of procedure calls. Construct_PFDF() uses the algorithm of Lengauer and Tarjan [3] to generate the PFDF. Then, it augments the PFDF with additional information. If the PFDF's root has an out-degree of 1 (keeping in mind that Stafford's PFDF uses strict dominance), the forest has no halts, and is thus a tree. If the PFDF's root has more than one edge extending from it, the PFDF's T/F type will be denoted as a forest. Stafford keeps track of the forest roots and then calculates all potential forward dominance (pfdom) arcs by reversing all the interrupted flow arcs of the source PCFG.

THE ALGORITHM: PROCEDURES: Construct_PCDG()

Weak control dependence analysis is a subset of the traditional type of "strong" control dependence analysis discussed here. Weak control dependence analysis recognizes that some loops may never break. As one may expect, weak dependence analysis is somewhat more complicated than strong control dependence analysis. Stafford's algorithm allows the user to choose between Ferrante's direct control dependence algorithm [1] and Podgurski's direct weak control dependence algorithm [5]. Upon receiving the user's choice, the correct algorithm executes. Next, all interrupted flow arcs from the procedure's corresponding PCFG convert into direct potential control dependence (dpcd) arcs in the PCDG, recognizing that the IFA targets have the potential to have the same control dependence as their parent nodes, pending return from the called procedure. All IFA targets lose any other dependencies that they may have.

After adding dpcd arcs, Stafford converts all arcs extending from the super entry vertex, V^I, into direct inherited control dependence (dicd) arcs. Then, for all IFA targets, she extends dicd arcs from those targets to every node that postdominates it, up to the root of the tree, or up to (but not including) the next IFA target. Stafford also detects all loop headers and extends dpcd arcs to every node that postdominates it, up to the root of the tree, or up to (but not including) the next IFA target.

THE ALGORITHM: PROCEDURES: Construct_CCDG()

The very first action that Construct_CCDG() takes is to connect all prior PCDG graphs with dicd arcs from call sites to their corresponding procedures' super entry vertices. Then, it contracts the call graph which, as previously stated, allows one to iterate through the call graph, even if the call graph models recursion. Additionally, the path used in contracting can be reversed for the post-order traversal that follows in the next step.

The next step is a process called "pfdom resolution," which either converts pfdom arcs with corresponding IFAs to indirect forward dominance arcs, or deletes them, if the called procedure referenced in the IFA is a tree or a forest, respectively. Until stated, the algorithm processes through each procedure in a postorder traversal. The reason for the postorder traversal is that if a source procedure's T/F type is found to be a forest, each procedure calling it must be marked as a forest, too, to indicate the embedded halt. When moving in a postorder traversal through the call graph, none of the procedures calling it have been processed, yet; when one processes a forest procedure, one can simply update the procedures that call it. This is a much more efficient solution than what would be achieved through an inorder traversal.

For every node processed, check if that node is involved with recursion. If not, and if the node's corresponding procedure is a forest, then remove the pfdom arcs corresponding to this procedure and notate the calling procedure's PFDF as a forest. (Note that in the implementation, one must keep track of the deleted arcs for future use.) If the node is not involved with recursion, and if the node's corresponding procedure is a tree, then all pfdom arcs corresponding to this procedure are converted into indirect forward dominance arcs (which will be kept for future use). (Nothing is done to the T/F type of the calling procedures if the procedure is found to be a tree.)

The call graph contract() routine will group nodes that are reachable from each other. In cases where a node is involved with recursion, one must process through each node group in the same manner as above. It is necessary to process the grouped nodes because they are all reachable from each other.

Once all postorder iteration of the call graph is performed, it is necessary to iterate, one last time, through each source procedure. Upon completion of this iteration, the CCDG algorithm is complete.

For each interrupted flow arc in the current procedure, check for arcs removed during pfdom resolution. Recall that these arcs were removed as a result of the procedure having embedded halts. For each corresponding target procedure, draw a dicd arc from the return vertex of that procedure to the original call site's IFA target. Doing this asserts that the IFA target of a procedure call with an embedded halt has the same control dependence as the return vertex of the called procedure. For IFA targets in loops, one must also remove the dpcd arcs incident to their call sites and make them direct control dependence arcs (i.e. "normal" arcs).

While iterating through the IFAs for the current procedure, check for indirect forward dominance arcs. Said arcs will have a dicd arc drawn in the opposite direction of the infdom arc. Such an arc will "look" like an IFA in a PCFG, in that it will have the same nodes, but while the IFA pertains to the CFG, the dicd arc pertains to the CDG, and such arcs only exist for procedures that call procedures that are of "tree" T/F type. (In cases where the called procedure is a forest, the dicd arc extends from the called procedure's return vertex to the IFA target. This step was outlined in the previous paragraph.) In cases where the IFA exists in a loop, remove all dpcd arcs to the call site.

Finally, remove the dpcd arc corresponding to the IFA. Upon removal of the last dpcd arc of the last procedure, the algorithm is completed. When the algorithm is completed, no more dpcd arcs should exist. The final CCDG consists of direct control dependence arcs ("normal" arcs) and direct inherited control dependence (dicd) arcs only.

THE IMPLEMENTATION

As earlier stated, the project implements Stafford's algorithm using SUIF (Stanford University Intermediate Format) and MachSUIF (Machine SUIF) as APIs. It assumes an input of a set of procedure CFGs in MachSUIF's .cfg format, and it produces a CCDG and all intermediary data structures in plaintext format simply as node pairs (e.g. "(main, A)").

This section will describe the software that comprises the heart of the project itself, and will explain the reasoning behind design decisions. It will also document the split between the MachSUIF API and the project code. The code, itself, was documented using Doxygen, an inline documentation engine similar to Javadoc. As of this writing, Doxygen is capable of generating documentation in HTML, RTF, or LaTeX. An HTML version of Doxygen's generated documentation is available on the Web at: http://www.cs.tufts.edu/r/serg/ccdg/doxygen/.

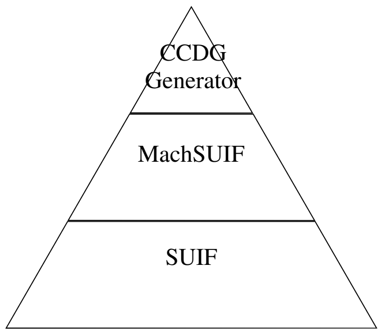

Figure 4. MachSUIF's relation to SUIF and the CCDG generator.

THE IMPLEMENTATION: API'S USED

MachSUIF and SUIF were the only two APIs used in the implementation. SUIF stands for "Stanford University Intermediate Format", and is a compiler designed for research purposes. Machine SUIF sits on top of SUIF, and the CCDG generator sits on top of MachSUIF, as shown in Figure 4. As diagrammed, there is no interaction between the CCDG generator and SUIF, itself. Though Stafford's algorithm is language-independent, and will work with most imperative languages, MachSUIF only handles C and FORTRAN inputs. Consequently, the CCDG generator only handles C and

FORTRAN inputs. Unfortunately, even if MachSUIF could handle C++, the CCDG tool would not be able to, as Stafford's algorithm does not address exception handling.

As an aside, because MachSUIF supports C, and the only code tested upon the CCDG generator was C code, all "source procedures", as they were previously called, will now be referred to as "source functions". Additionally, all project code documentation and commenting refers to "functions", not "procedures."

MachSUIF facilitates the process of designing compiler passes that optimize machine code. The process of generating a CCDG is actually a compiler pass, and is controlled by a master class, CcdgSuifPass, which inherits from a SUIF class, PipelinablePass. According to SUIF's documentation [7], a pass is pipelinable if its computation can be applied to each source function independently. Most of SUIF's passes work through implicit iteration. When one sets up a pipelinable pass subclass, SUIF iterates through all functions, calling do_procedure_definition() once per each source function.

Before discussing the issues involved in the implementation, we will provide documentation on installing SUIF and MachSUIF in Tufts University's Computer Science Department Solaris environment. Note that while Doxygen is not an API, it was used in generating documentation for the CCDG generator code base. Were someone to update the project's documentation, one would need to check for this installation. At the time of this writing, Doxygen was available on most Tufts University CS Department Linux systems.

THE IMPLEMENTATION: INSTALLATION

The version of SUIF used for this installation is SUIF 2.2.0-4 and the version of MachSUIF used was 2.02.07.15. At the time of implementation, these were the most recent versions of SUIF and MachSUIF available. SUIF and MachSUIF were compiled with gcc 2.95.3 on a Sun Ultra-250 running Solaris 5.9. It may be possible to use gcc 3.*x* to compile SUIF, however I have not tested my project with anything above 2.95.3, as SUIF's README mentioned being tested with 2.95.3.

The following are the steps taken to install the current version of SUIF and MachSUIF found on Tufts CS systems:

1. Ensure that proper directory and umask for install are in place.
2. Untar basesuif (basesuif-2.2.0-4.tar.gz).
3. Untar MachSUIF (machsuif-2.02.07.15.tar.gz).
4. Move basesuif's results to the appropriate directory.
5. Set appropriate environment variables.
    a. Execute `/bin/sh ./install`
6. Add MachSUIF to the list of packages to install (so that one does not have to make a separate install of MachSUIF).
    a. Move the result of the untarring to $NCIHOME/suif/suif2b/machsuif
    b. Execute `setenv $MACHSUIFHOME $NCIHOME/suif/suif2b/machsuif`
    c. Execute `echo machsuif >> $NCIHOME/suif/suif2b/extra_suif_packages`
7. Patch MachSUIF's bit_vector.cpp file.
    a. Execute `patch -p1 < $MACHSUIFHOME/patches/basesuif-2.2.0-4.patch`
8. Fix a glitch with gcc 2.95.3's templates support and MachSUIF.
    a. Execute `echo EXTRA_CXXFLAGS += -D__STL_USE_SGI_ALLOCATORS >> $NCIHOME/Makefile.options.gcc`
9. Compile.
    a. Execute `make setup`
    b. Execute `make`
10. Test.
    a. Execute `make test`
11. Clean up the MachSUIF installation directory.
    a. Execute `cd $MACHSUIFHOME; make tidy`
12. Install the Solaris front end for SUIF.
    a. Because the file does not extract into its own directory, create a new directory: Execute `mkdir foo; cp Sparc_Solarisc2s.tar.gz foo; cd foo`

    b. Extract the archive file.
    c. Execute `make install`

13. Remove all unnecessary archives and the temporary directory created in step 12.a.

The CCDG generator can run in three different modes: normal mode, file mode, and debug mode. In normal mode, the program will write all output to standard out. In file mode, all output will be to files. A file named <*prefix*>.cfg with a source function named <*functionname*> will result in files named:

- <*prefix*>_<*functionname*>.pcdg
- <*prefix*>_<*functionname*>.pcfg
- <*prefix*>_<*functionname*>.pfdf
- <*prefix*>.cg, and
- <*prefix*>.ccdg.

To compile the actual CCDG project, first choose the mode in which to run the generator, and set it as a `#define` in ccdg.h. Locate the line:

```
#define RUNMODE <MODE>
```

and change `<MODE>` to one of the following:

- `DEBUGMODE`
- `FILEMODE`
- `NORMALMODE`

simply `cd` to the directory containing the project (at the time of this writing, /r/serg/ccdg/) and type `make` to compile. That will create an executable file in $NCIHOME/bin called do_ccdg. To setup SUIF for running, execute `source /r/serg/suif.login`. (One may wish to source it, automatically, in your $HOME/.login file.) C code must be run through preprocessing prior to running it with do_ccdg. A small Perl script contained with the project called c2cfg will run all the necessary compiler passes to get a .c file into proper

.cfg form. (You may want to put this file in $NCIHOME/bin.) Upon generating the .cfg file, <*prefix*>.cfg, run `do_ccdg <prefix>.cfg`.

THE IMPLEMENTATION: DATA STRUCTURES

Only one-fourth of the code of the CCDG project is involved with directly implementing Stafford's algorithm. The remainder of the code is either involved with setting up the MachSUIF pass or providing robust storage mechanisms with ample and easy means of iterating through and retrieving items from said mechanisms. A full run-down of the available data structures, complete with inheritance hierarchies, is available in the Doxygen-generated developer documentation. However, a quick look at some of these data structures will reveal much about the code's underlying details.

THE IMPLEMENTATION: DATA STRUCTURES: Edges

Most of Stafford's algorithm involves iterating through data structures, adding and removing different kinds of edges, and converting edges to different types. Thus, it is important that any supporting data structures facilitate this as much as possible. My philosophy in coding the project was to move all the internal details of the supporting data structures away from the algorithm implementation, within reason, so that there would not be a huge gap between the written Stafford algorithm and the actual implementation. Edge collections were stored as arrays, just as they were in Stafford's algorithm. The primary hope was that bugs resulting from incongruity with the written algorithm would become more apparent.

In the implementation, every graph is described as a set of edges, not as a singular graph data structure. There is a different edge object instance for every type of edge, e.g. IFA, dicd, or pfdom arc. Most edges, regardless of type, use the same data structure, CfgEdge. Despite the name, CfgEdge data structures are not just used in the PCFG implementation, but are used to describe edges in the PFDF, PCDG, and CCDG implementations, as well. The reason for the name is that the edge contains CfgNode objects. Though the many supporting graphs used in the algorithm are different, the nodes are the same throughout. For example, node 3 in a procedure's CFG will be the same node 3 in the procedure's FDF and the same node in the procedure's CDG. So, the name CfgEdge does not describe what type of graph the edge belongs to, but rather, what type of node it contains. There is a second type of edge, a CgEdge, which holds call graph nodes of type CgNode. The third and last type of edge, CfgSpanningEdge, holds CfgNode objects that span multiple source functions (CfgSpanningEdge inherits from CfgEdge). These edges all stay consistent with the naming convention.

Every edge, (u, v), has two vertices, u, and v. Thus, every edge object, whether CfgEdge, CfgSpanningEdge, or CgEdge, has methods u() and v(), which return the respective nodes. In addition to holding CgNode objects, the CgEdge class holds CfgNode objects to act as call site and target markers, so in addition to providing u() and v() methods, it provides u_cfg() and v_cfg() methods, as well. Every edge also provides a function, fprint(), which prints the edge to a target file descriptor. Additionally, every edge overloads the == operator in order to provide more convenience to the user of the edge class.

THE IMPLEMENTATION: DATA STRUCTURES: Collections

As stated earlier, every graph in the implementation is a collection of edges. Every edge is of type CollectionItem. Every edge collection inherits from the class Collection, which stores items of type CollectionItem. Collections store items in arrays. The thought of implementing them as linked lists was entertained as a possibility, but due to the many different collections that a node could be in simultaneously, one could not put "next" and "last" pointers in the nodes themselves: one would have to wrap the nodes and store the list pointers in the wrapper class. Also, it was realized that most collections would be iterated through many times more than they would be added to, and arrays are faster than linked lists in terms of iteration speed. So, I decided that the best approach would be with arrays.

```
void Collection::add(CollectionItem *ci) {
  CollectionItem **temp;
  int i;

  if(num_items >= max_items) {
    max_items += COLLECTION_CHUNK_SIZE;
    temp = new CollectionItem*[num_items];
    for(i = 0; i < num_items; ++i)
      temp[i] = set[i];
    delete set;
    set = new CollectionItem*[max_items];
    for(i = 0; i < num_items; ++i)
      set[i] = temp[i];
  }
  set[num_items++] = ci;
}
```

Figure 5. Code for the Collection base class.

Unfortunately, arrays in C are not dynamically expandable. Thus, the internal code for adding items to the array is not very elegant. As one can see from Figure 5, the Collection's storage array has to be re-allocated after adding a certain number of items, as defined by COLLECTION_CHUNK_SIZE. I decided that time is more important than disk space, and have therefore set COLLECTION_CHUNK_SIZE to a reasonably high value: 20. I estimate that most Collections will never exceed a size of 20, and even fewer would exceed a size of 40.

There is another unfortunate condition: one collection type does not inherit from the Collection base class. The class CfgNodes, which holds a collection of type CfgNode, is incapable of inheriting from type Collection because the type CfgNode is provided by MachSUIF, and is therefore incapable of inheriting from type CollectionItem. The thought of wrapping the class appealed to me, but I have shied away from wrapping any classes in this project, in order to achieve speed.

All Collections support the method fprint(), which prints all of its CollectionItems to a file descriptor. In each case, the fprint() routine simply iterates through its CollectionItems and calls its objects' own fprint() methods.

THE IMPLEMENTATION: DATA STRUCTURES: Call graph

Occupying roughly one-third of the number of lines of code in the CCDG project, the call graph class and its supporting data structures make up, by far, the single largest and most complicated component in the implementation. At its simplest, the call graph is simply a pointer to a root with each node holding information about its own children. But the call graph object also provides postorder and inorder iteration methods, search methods, and a contract() method implementing a simplified version of Gabow's path-based depth-first search algorithm [2] (see "STEPS TAKEN: Construct_CCDG()" for details on the simplification).

The CallGraph object points to a root of type CgNode. Each CgNode has children of type CgNode, stored in a Collection called CgNodes. CgNode objects can also be stored in a CollectionItem called CgEdge, and in a Collection called CgEdges.

CgEdges store more information than CgNodes: they also store the corresponding CfgNodes that contain the call site and target nodes corresponding to that edge. Ideally, instead of having CgNode children, each node would have a CgEdges collection of children. Unfortunately, gcc compilation fails when placing a CgEdges object as a member of a CgNode object. The cause for it is still unknown, but I suspect that either the two are circularly dependent on each other, or that gcc 2.*x* simply cannot support the combination of these two closely-related inherited classes. Consequently, the CallGraph class stores the CgEdges, and thus the CgNode children of a CgNode are only useful when initially contracting the graph.

THE IMPLEMENTATION: STEPS TAKEN

The CCDG generator had several issues during several parts of its development. All of the issues were resolved, though some solutions were more elegant than others. This section will provide a run-through of the issues encountered with each portion of the algorithm.

THE IMPLEMENTATION: STEPS TAKEN: Construct_PFDF()

On lines 1 – 3 of Stafford's Construct_PFDF() procedure, the algorithm builds the data structure A^T for use with Lengauer and Tarjan's PFDF algorithm [3]. A^T provides all "return" or "halt" edges. Because MachSUIF provides forward dominance information out-of-the-box, the construction of A^T is not necessary. Nor is the Forest_list data structure necessary. However, we compute both, regardless.

In Construct_PFDF(), the Cfg generated by MachSUIF does not completely match the PCFG that Stafford specifies. This is partly because MachSUIF generates a control flow graph for machine code instructions instead of for program statements.

Also, in Stafford's examples, a no-op node follows each procedure call. Some augmentation of the default Cfg data structure follows its generation, in order to fit it to Stafford's PCFG. In most cases, the match is not identical, but it is close. The biggest problem is that Stafford prefers a 1:1 correspondence between program statement and PCFG node. I have not yet figured out a way to make that happen.

On lines 6 – 9 of Construct_PFDF(), one checks the out-degree of the root of the PFDF and assigns the T/F type based on that out-degree. First, it is important to know that MachSUIF does not use strict dominance, and as a result, one would have to check that the out-degree of the PFDF root equals 2, not 1. But in actuality, one cannot use that qualification, either, because MachSUIF treats calls to exit() in source code as function calls, not as halts. Thus, the exit() call site in a PCFG has a vertex that extends from it to a child node, just as does every other function call. Consequently, the PFDF root never has an out-degree of more than 2.

Finally, on line 14 of Construct_PFDF(), Stafford's algorithm assumes that all interrupted flow arcs have been calculated with the PCFG. MachSUIF does not provide IFAs, so one must calculate them there. (As the IFA is an invention of Stafford's algorithm, one would not expect any API to provide them, outright.)

THE IMPLEMENTATION: STEPS TAKEN: Construct_PCDG()

On lines 1 – 6 of Construct_PCDG(), Stafford's algorithm assumes that a decision has been made about strong or weak dependence analysis. In the implementation, the user specifies strong or weak dependence analysis in a function called initialize(), which runs prior to any processing of source functions. I made the decision that the same type of analysis must be performed on all source functions. While choosing different analyses for different source functions will not cause problems in the algorithm, it does cause problems with a command line interface-based tool that must prompt the user once for each source function in the code; when dealing with large software with dozens of function calls, this might be quite a nuisance. In a GUI-based CCDG tool, one may decide make a default value and give the user the ability to change analyses for separate functions.

On line 2 of Construct_PCDG(), Stafford's algorithm calls for the computation of Ferrante, et al.'s direct control dependence algorithm [1]. In implementing Ferrante's algorithm, I stopped short of implementing region nodes, as I do not see them as being necessary in Stafford's algorithm. Additionally, in my calculation of Ferrante, there is no edge between (V^I, V^R), though there should be. So, I put one in manually.

On line 5 of Construct_PCDG(), Stafford's algorithm calls for the computation of Podgurski's direct weak control dependence algorithm [5]. This step remains un-implemented, due to the shear work involved in understanding and implementing a second large-scale algorithm, with all of its necessary supporting objects.

On line 11 of Construct_PCDG(), Stafford adds V^I to the collection of nodes, T, which will be used in processing beginning on line 15. That step was put in place so that lines 15 through 22 will add dicd arcs extending from every PCDG initial vertex to its corresponding children. This works under the assumption that V^I is always a child of node 1 in the PFDF. In MachSUIF's PFDF, V^I is always a child of V^R. Consequently, this does not work, so I have created my own steps to ensure the same end, and thus T does not contain V^I in the implementation.

On line 18 of Construct_PCDG(), the variable s refers to the call site adjacent from the same node, t, referenced in lines 15 and 16. In other words, s and t in lines 15 – 23 in Construct_PCDG() represent the (s, t) in I_G from line 12.

On line 25 of Construct_PCDG(), one must insert dpcd arcs, not dicd arcs, as one is instructed to do so. This is a typo in the Stafford algorithm.

THE IMPLEMENTATION: STEPS TAKEN: Construct_CCDG()

Until this point in the algorithm, no arcs have spanned any source functions. Previously, all procedure-specific data structures were arrays, and each source function had a corresponding array index number. Line 3 of Construct_CCDG() calls for the first set of spanning arcs to be created. At this point, new arcs go into a different data structure than before. Before the end of the procedure, all necessary procedure-specific dicd arcs will also be copied into the same data structure that the arcs from line 3 use.

On line 6, the call graph contract() routine, originally specified by Gabow [2], is actually a simplified version of the Gabow algorithm. In short, the new algorithm performs a depth-first search and keeps track of the current path with a stack. Path information is stored in the nodes, themselves, and the stack grows and shrinks as the DFS routine recurses through the source functions. Upon hitting a node already in the path stack, it contracts all nodes from that node to the end of the stack. I see no reason why this simplified algorithm is not as appropriate as the original Gabow algorithm.

Lines 24 – 42 of Construct_CCDG() remain un-implemented. While most of the lines are the exact same as from lines 10 – 23, I have chosen not to implement them to save time, testing, and the possibility of infinite looping if I do not handle recursion correctly. Additionally, at this point, my call graph data structure has no set of methods to report back which nodes are grouped with which. I do not see it as a difficult task to put this functionality into place, but it has not been done, yet.

THE IMPLEMENTATION: THE PROJECT AS A COMPONENT

Although it was not designed as such, the CCDG generator project can be thought of as a reusable software component, taking in an input of a CFG for the source code, and producing a CCDG, all PCFGs, PFDFs, PCDGs, and the program call graph. This section attempts to draw lines between MachSUIF and the CCDG generator code.

Unfortunately, MachSUIF is not entirely object oriented. Methods related to postdominance information are contained in a DominanceInfo object. However, extracting information from the Cfg object or CfgNode objects requires ordinary functions, not methods. A partial alphabetical listing of used MachSUIF methods and functions is available in Table 1, below (documentation for each of these is available through MachSUIF's documentation page [4]).

Parent Object	Function/Method
-	AnyBody* get_body(OptUnit*)
	CfgNodeHandle nodes_end(Cfg*)
	CfgNodeHandle nodes_start(Cfg*);
-	CfgNode* get_entry_node(Cfg*)
-	CfgNode* get_exit_node(Cfg*)
-	CfgNode* get_succ(CfgNode*, int pos)
-	CfgNode* insert_empty_node(Cfg*, CfgNode *tail, CfgNode *head)
-	IdString get_name(SymbolTableObject*)
-	Instr* get_cti(CfgNode*)
-	Sym* get_target(Instr *instr)
-	bool ends_in_call(CfgNode*)
-	bool optimize_jumps(Cfg*)
-	bool remove_unreachable_nodes(Cfg*)
	bool postdominates(CfgNode *dominator, CfgNode *dominatee) const
-	int get_number(CfgNode *)
-	void canonicalize(Cfg*, bool keep_layout = false, bool break_at_call = false, bool break_at_instr = false)
-	void fprint(FILE*, Cfg*, bool follow_layout, bool show_code)
DominanceInfo	CfgNode *immed_postdom(CfgNode *n) const
DominanceInfo	void find_dominators()

Table 1. Functions and methods provided by MachSUIF that were used in the CCDG generator. Note that this does not include all procedures used in setting up the MachSUIF pass.

The CCDG generator code actually uses more functions and methods than that, but the ones omitted serve the sole purpose of setting up the MachSUIF compiler pass that generates the CCDG and all supporting data structures. All non-pass-related data structures used are available in Table 2, below.

Object Type	Object Name
class	Anybody
class	Cfg
typedef list<CfgNode*>::iterator	CfgNodeHandle
class	CfgNode
class	DominanceInfo
class	Instr
class	IdString
typedef ProcedureDefinition	OptUnit
typedef Symbol	Sym

Table 2. Objects provided by MachSUIF that were used in the CCDG generator implementation. Note that this list does not include all objects used in setting up the MachSUIF pass.

Unfortunately, there was no thought put into swapping out MachSUIF during most of the development of the project. Had there been, MachSUIF's functionality would have been wrapped wherever possible, so that every reference to and instance of the above would have to be replaced once, versus the many times in which they were used. Unfortunately, that did not happen, and as a result, separating MachSUIF and the CCDG generator would be a nontrivial task.

CONCLUSION

On the whole, this project was a success. The generator is very easy to use, requiring minimal user interaction, and the output is human-readable. The two biggest problems were that it was not completed in its entirety, and that MachSUIF was not a perfectly appropriate fit for the project. See "Future Work" for additional information on remedying these problems.

CONCLUSION: FUTURE WORK

Due to the extensive nature of this project, and the learning curve necessary to understand control dependence analysis, the entirety of Stafford's algorithm was not fully completed in code. Most prominently missing is the implementation of Podgurski's weak control dependence algorithm [5], and support for recursion (lines 24 through 43 in Stafford's Construct_CCDG() procedure). The remainder of the code for Stafford's algorithm has been implemented in full.

Perhaps an even more severe limitation of the project is that it will not accept more than one source .cfg file. Consequently, that prohibits analysis of programs of any significant size, which will all span multiple files. MachSUIF's documentation mentions that one must use its scripting capabilities to put several CFGs into one file_set_block. From what it seems, no changes must be made to the CCDG pass code, but rather to the preprocessing that converts a .c file into a .cfg file. If one could put multiple .c files into a .cfg file, that would solve the multiple source files problem.

An even more ambitious project would be to remove MachSUIF as an API and replace it with a more appropriate API. The problem with MachSUIF is that it deals with machine code, and provides a CFG (and therefore a PFDF) for machine code. There is rarely a 1:1 correspondence between program statements and machine code, and in many cases, there will not be a 1:1 correspondence between program statements and CFG nodes. This is an unfortunate problem for anyone wanting to use the CCDG tool for serious development or maintenance purposes. Another point of failure was that MachSUIF treated embedded halts as ordinary function calls, in that embedded halt nodes still had children, and there was no way to remove the edges incident to those children. A good, appropriate API will be one that:

- provides a PCFG for source code, rather than machine code
- provides a 1:1 correspondence between source code statements and nodes
- can generate a PFDF from a PCFG (otherwise one will have to implement Lengauer and Tarjan [3] manually, as MachSUIF provided a PFDF)
- provides "join" nodes after branches resulting from "if" statements (MachSUIF does provide this)
- provides the ability to add nodes (MachSUIF does provide this)
- either provides no CFG node children after an embedded halt, or provides the capability to remove edges

It may be possible that SUIF is an appropriate API for this project; however I find SUIF to be so poorly documented and supported that it may be a poor choice for someone without a sense of adventure. To emphasize, I will add the otherwise implicit qualification for an appropriate API:

- has documentation that is thorough and well-organized, and well-supported

Another ambitious project would be to reuse more code in the CCDG generator through the use of templates. Currently, a strong inheritance hierarchy exists for collections of objects, and every edge and call graph node inherits from the type CollectionItem, which is the only type that a Collection object can hold. Unfortunately, CfgNode objects cannot be held in a collection because they are a built-in MachSUIF type, thus they cannot inherit from anything. However, with templates one could implement a Collection that held anything, regardless of data type. Prior to re-implementing with templates, one would have to recompile SUIF, MachSUIF, and the CCDG generator with gcc 3.*x*. All attempts at combining templates with inheritance in gcc 2.*x* have failed. So I have been told, many issues with templates and inheritance have been fixed with gcc 3.*x*.

On the topic of C/C++ issues, very little regard is paid to memory deallocation in the CCDG generator. The reason for this is that many nodes and edges are passed around, spanning functions and collections. In order to properly deallocate the memory used by edges and nodes would require extensive and complex reference counting. With algorithm correctness being of a higher priority than memory management, and because the program simply runs from start to finish with little or no user interaction, I have opted not to do that. I do free some memory where it is obvious to do so, but never more than that. A future project, if needed, would be to count references and deallocate edges and nodes when no longer necessary.

Perhaps the most obvious piece of future work would be a graphical user interface for the CCDG tool. The project in its current state provides edges as node pairs, e.g. "(main, A)" with optional notations afterward, e.g. "(main, A) dicd". Reconstructing a graph has the potential to be a significant task if the source program is sufficiently large. As of this writing, there is one GUI project underway (see "Related Work").

CONCLUSION: RELATED WORK

There are two related projects reaching conclusion at the same time as this project. Currently, Shuanghong He is developing a front end to this project, using the SUIF/MachSUIF installation with my project as a TCP/IP server to talk to a client running a Java front end. My project, by itself, provides graphs as text files describing sets of edges. He's project parses those files and converts the edges into AT&T's dot format and image files. That saves the often cumbersome step of translating the edges into an actual pictorial representation. Matthew Salter is finishing development of a component architecture description language, Simple Architecture Definition Language ($ADL), which identifies control dependencies between components based on the $ADL specification. Salter's project analyzes inter-component relationships, whereas this project analyzes intra-component pathways. One could attain a full picture of a component assembly's control dependence with a combination of Salter's project and this.

REFERENCES

1. J. Ferrante, K.J. Ottenstein, and J.D. Warren. *The program dependence graph and its use in optimization.* ACM Transactions on Programming Languages and Systems, 9(3):319--349,October 1987.

2. H. N. Gabow. *Path-based depth-first search for strong and biconnected components.* Information Processing Letters 74, 2000, Pages 107--114.

3. T. Lengauer and R.E. Tarjan. *A fast algorithm for finding dominators in a flowgraph.* ACM Transactions on Programming Languages and Systems, 1(1):121--141, July 1979.

4. G. Holloway and M. D. Smith. *The Machine-SUIF Machine Library.* The Machine-SUIF documentation set, Harvard University, 2000. http://www.eecs.harvard.edu/hube/software/

5. H.A. Podgurski. The Significance of Program Dependences for Software Testing, Debugging, and Maintenance. PhD thesis, University of Massachusetts, Amherst, Massachusetts, September 1989.

6. J.A. Stafford, "A Formal, Language-Independent, and Compositional Approach to Control Dependence Analysis," Ph.D. Thesis, University of Colorado, July 2000.

7. Unknown. *The Suif 2 Compiler System.* SUIF 2, Stanford University, circa 1999. http://suif.stanford.edu/suif/suif2/