Context-Sensitive Control Flow Graph

Jim-Carl Ng

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation
Ng, Jim-Carl, "Context-Sensitive Control Flow Graph" (2004). Retrospective Theses and Dissertations. 20215.
https://lib.dr.iastate.edu/rtd/20215

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Context-Sensitive Control Flow Graph

by

Jim-Carl Ng

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Suraj C. Kothari, Major Professor
Simanta Mitra
Daniel Berleant

Iowa State University
Ames, Iowa
2004
Copyright © Jim-Carl Ng, 2004. All rights reserved.
This is to certify that the master's thesis of

Jim-Carl Ng

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
# TABLE OF CONTENTS

**LIST OF FIGURES** ....................................................... v

**ACKNOWLEDGEMENTS** .................................................... vii

**ABSTRACT** ................................................................. viii

**CHAPTER 1. Introduction** ............................................... 1

**CHAPTER 2. Literature Survey** ......................................... 3

2.1 Visualizing Control Flow ............................................ 3

2.1.1 Visualizing Interprocedural Control Flow ....................... 4

2.2 User's Concern Approach in Software Analysis .................... 5

2.3 Representing Concerns in Graphs .................................... 6

**CHAPTER 3. Context-Sensitive Control Flow Graph** ................. 8

3.1 Complexity of Graphs ................................................. 8

3.2 Marking .................................................................. 8

3.3 Relevance of Program Entities ...................................... 10

3.4 Intra-procedural Context-Sensitive CFG .......................... 10

3.5 Inter-procedural Context-Sensitive CFG ......................... 13

3.6 Context Sensitive Call Order Graph ................................ 13

3.6.1 Context-Sensitive Node Deletion ................................ 15

**CHAPTER 4. ContextCFG tool and its Implementation** .......... 17

4.1 Overview ................................................................. 17
4.2 Preprocess Work ......................................................... 17
  4.2.1 eXtensible Common Intermediate Language (XCIL) .............. 18
  4.2.2 Call Order Analysis .................................................. 19
4.3 Markings ...................................................................... 20
4.4 Graph Construction Module .............................................. 21
  4.4.1 Building Context-Sensitive Call Order Graph ...................... 22
  4.4.2 Building Intra-procedural Context-Sensitive CFG .................. 24
  4.4.3 Building Inter-procedural Context-Sensitive CFG .................. 26
4.5 Visualization module ...................................................... 26
  4.5.1 Procedure on using ContextCFG tool ................................ 27

CHAPTER 5. Results ................................................................. 29
  5.1 Graphical Results .......................................................... 29
    5.1.1 Intra-procedural level ............................................... 29
    5.1.2 Inter-procedural level ............................................... 29
  5.2 Quantitative Results ....................................................... 34

CHAPTER 6. Program Path Searching Queries .............................. 38
  6.1 Implementing Program Path Searching Queries ........................ 39
    6.1.1 Program Path Search from a start marking to a set of end markings 39
    6.1.2 Program Path Search from a start marking to end of method without encountering a set of end markings ...................... 39
  6.2 Program Path Searching Queries Results ............................... 40
  6.3 Future work on Program Path Searching Queries ...................... 40
  6.4 Related Work ................................................................ 43

BIBLIOGRAPHY .................................................................. 44
# LIST OF FIGURES

<p>| Figure 3.1 | A sample CFG of method ipproc() in XINU operating system | 9 |
| Figure 3.2 | A simple CFG for an if-else statement | 11 |
| Figure 3.3 | A method in C language and the constructed Context-Sensitive CFG | 12 |
| Figure 3.4 | Markings affecting statements in other methods | 14 |
| Figure 3.5 | Figure showing what are being retained in Context-Sensitive Call Order Graph | 15 |
| Figure 3.6 | Context-Sensitive node deletion | 16 |
| Figure 4.1 | The architecture of building Context-Sensitive Control Flow Graph | 18 |
| Figure 4.2 | Conversion from C code to XCIL | 19 |
| Figure 4.3 | Sample C code and its corresponding XCIL data | 20 |
| Figure 4.4 | The structure of callorder.xml | 21 |
| Figure 4.5 | Generation of markings XML file | 21 |
| Figure 4.6 | Recursive function traverses callorder.xml’s JDOM tree to create Call Order Graph | 23 |
| Figure 4.7 | The Call Order Graph Node class and its fields | 23 |
| Figure 4.8 | The Context Sensitive CFG Node class and its fields | 25 |
| Figure 4.9 | Screenshot of ContextCFG tool | 27 |
| Figure 5.1 | Ordinary CFG for ipproc() | 30 |</p>
<table>
<thead>
<tr>
<th>Figure 5.2</th>
<th>Context-Sensitive CFG for ipproc() with getbuf() and freebuf() as markings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 5.3</td>
<td>Call Order Graph with ipproc() as root method</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Context-Sensitive Call Order Graph with ipproc() as root method, and getbuf() and freebuf() as markings</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>Call Order Graph with ipproc() as root method and panic() deleted</td>
</tr>
<tr>
<td>Figure 5.6</td>
<td>Context-Sensitive Call Order Graph with ipproc() as root method and panic() deleted context-sensitively</td>
</tr>
<tr>
<td>Figure 5.7</td>
<td>Total nodes and edges count comparison between ordinary CFG and Context-Sensitive CFG</td>
</tr>
<tr>
<td>Figure 5.8</td>
<td>Formula for calculating reduction percentage</td>
</tr>
<tr>
<td>Figure 5.9</td>
<td>Total nodes and edges count comparison between ordinary Call Order Graph and Context-Sensitive Call Order Graph</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>A sample code from XINU operating system for querying</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Query 1</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>Query 2</td>
</tr>
</tbody>
</table>
I would like to take this opportunity to thank Dr. Suraj Kothari for his guidance, patience and support throughout this research and the writing of this thesis. He inspires and guides me into this thesis, and has shown me how the actual engineering world works. Special thanks to Jon Mathews, also a KCS lab member, who has been a teacher and a friend throughout my 2-years graduate studies. Because his research uses my research results, he has helped a lot in guiding my research direction. He gave a lot of valuable suggestions and inspirations that helped solve a lot of problems in this thesis. I like to thank my parents for supporting me throughout my years of education, both mentally and financially. For all this years they have been my model to look up to. They inspire me in all ways, and they are the greatest gift of my life.

I also like to express my thanks to all the KCS lab members who helped me with various aspects of conducting research and the writing of this thesis.
ABSTRACT

Control Flow Graph (CFG) is known to be essential in compiler optimizations, and quite useful in program comprehension. But visualizing and understanding CFG is hard because the CFG is often too large. The problem becomes harder if one were to understand inter-procedural CFG. Often the user wants to understand specific aspects of the CFG. Hence omitting parts of CFG irrelevant to the user’s current interest becomes a desirable approach in working with CFG.

We formalize an approach of constructing and visualizing CFG that takes users’ interest into account. Users have to specify the program artifacts that are of interest to them, then we define the Context-Sensitive CFG that is relevant to the artifacts. The Context-Sensitive CFG is much smaller in size. The actual compression depends on the context and its spread in the given code. We show some results using the XINU operating system code as the test case.

While the Context-Sensitive CFG captures only the relevant details, it could still be of substantial size and complexity. We propose query capability as a next step to help the user by extracting details related to a specific question. For example, the user can apply a query to check if there is an execution path in the CFG along which there is a missing memory deallocation and thus a possible memory leak. This thesis only implements queries that operate only within the scope of a method (intra-procedural).

We have built a proof-of-concept tool that allows the user to visualize Context-Sensitive CFG and make queries.
CHAPTER 1. Introduction

A Control Flow Graph (CFG) represents the control flow in source code. More formally defined, a control flow graph is a directed graph that consists of nodes and edges, in which nodes represents single statements and edges represents control flow between the nodes ([21]). By looking at a node in the control flow graph, and following its flow, helps a programmer to understand the program.

The CFG is important in compiler code optimization, static analyses and program comprehension. It is not surprising that it has been widely studied. Visualizing CFG is hard because the CFG is often too large. The problem becomes harder if one were to visualize inter-procedural CFG.

We propose a way of constructing control flow graph, in which the graph only shows information that is relevant to users’ interests, and compressing the other information so that it is still available to users upon request. We let users decide what are the program points in the source code they are interested in, and build the interprocedural control flow graph that captures the program points. Other nodes in the control flow graph get collapsed into block nodes according to certain rules. These rules will be discuss in details in Chapter 3. We named this new control flow graph as Context-Sensitive Control Flow Graph.

Do note that the term Context-Sensitive for our Control Flow Graph is different from the term context-sensitive as used in static analyses. In static analysis, Context-Sensitive means the analysis results are dependent on different contexts of the call. Here, Context-Sensitive means the CFG result is dependant on the user-specified context.
In the following chapter, we first do a survey on literature of related work. Chapter 3 discusses about issues related to building the Context-Sensitive Control Flow Graph, while Chapter 4 shows the implementation details of it. Finally, Chapter 5 gives the results.
CHAPTER 2. Literature Survey

As mentioned in previously, a Context-Sensitive CFG visualizes a CFG by hiding out information irrelevant to the user's interest. To conduct a literature survey on this thesis, we can mainly divide it into 2 aspects: visualizing control flow and user's concern-based approach in software analysis.

2.1 Visualizing Control Flow

A Control Flow Graph (CFG) of a procedure (a method in a program) is defined as a directed graph that consists of nodes and edges([21]). The nodes represent single statements of the procedure while the edges indicate the possible control flow among the nodes. Since CFGs presents program flow in a graph view, they have been long promoted by programmers as extremely useful program comprehension aids. However, the difficulties of visualizing CFGs have always been a hard problem to visual designers. A function consists of 10 to 20 statements can result in a CFG too big to comprehend.

Visualizing control flow does provides a lot of advantages. It mainly helps in program understanding, and hence help to reduce maintenance by showing the complexity of control flow. It can help to establish norms and standards in quality control programmes and to define testing methods. It can also provide better understanding and definition of the cohesion and coupling of software units.

Though a lot of work and researches has been done on control flow analysis, only several had touch base on the visualization of it. This is mainly because CFGs are usu-
ally read by programs and not human eyes. Since visualizing CFG possess graph size problem, researchers had look into different ways of visualizing it. [18], [19], [9] and [11] change the representation of CFG into a similar representation, by drawing control flow edges beside the statements in the code, thereby directly treating statements as nodes, without actually drawing the nodes. These approaches convey a lot of information within a limited space. [11] and [9] also allows a feature of collapsing control structure blocks, where users can hide out uninterested blocks of code. These approaches all provide more comprehensibility than traditional CFG. However, the CFGs are still very much incomprehensible when displaying more than 50 lines of code. This is because when user has to scroll across pages to follow the control flow, the graph is very much incomprehensible. To help with this situation, [11] and [9] both integrated their representations into software development IDEs, in which they provide an additional feature that shows the immediate parent control structure block of a selected statement, to help user to follow the control flow.

Despite the comprehensibility problem remains, these approaches only address the visualization of intra-procedural CFG. To surf across methods, programmers can only utilize a code explorer like window to browse through different files and methods. We intend to look into addressing transformation of control flow inter-procedurally. Hence in this thesis, we extend the idea of Context-Sensitive into call order graph. The ability of call order graph that tells user the caller and callee context of a method directly addresses the transition of control flow among methods.

2.1.1 Visualizing Interprocedural Control Flow

Most of the work done on interprocedural CFG maintains a repository of intra-procedural CFGs and hyperlink all call nodes to the callee method’s CFG ([24], [10], [14], [13]), instead of building one massive CFG that connects all intraprocedural CFGs together. This is due of the fact that intraprocedural CFGs are invariant towards calling
context. Although not all work done on are related to visualization (most of them are for static analyses), they all agree that hyperlinking call nodes to the callee method’s CFG provides more comprehensibility or efficiency.

A lot of the work done on CFG also computes a call order graph to provide a full view of interaction between methods in the system ([24], [10], [14]). Nodes in the call order graph link to the intraprocedural CFGs of the corresponding methods. A combination of call order graph and intraprocedural CFG can provides good navigation between intraprocedural CFGs. However, a call order graph for a large size software system is often a massive graph that is hard to comprehend. To visualize interprocedural CFG in a more comprehensive way, the call order graph size needs to be relatively small.

In this thesis, we build Context-Sensitive Call Order Graph to address the control flow information between methods that users are interested in.

2.2 User’s Concern Approach in Software Analysis

Utilizing user’s concern in software analysis is good approach to increase efficiency of analyses and filter out redundant work. Users often do have an aspect in mind that they would like to extract from source code to understand it better. Due to the fact that static software analyses often require high computational cost, user’s concern can help tools to determine which part of the input program is not required to be analyzed, therefore saving computational cost. Do note that program slicing concept for data flow analysis is not utilizing user’s concern because the analysis has to be performed on the entire input program, regardless of the slice that the user wants to acquire.

This approach requires the analysis program to be able to provide an interface for the user to extract the aspect he has in mind. This interface further requires a source code representation that can uniquely identify program entities like statements, conditions or even operations, depending on the granularity of the analysis.
Suraj Kothari’s work on the eXtensible Program Specification Language (XPSL) is a language that serves as a query language that extracts aspects out from source code ([6]). It utilizes the eXtensible Common Intermediate Language (XCIL) to uniquely identify all program entities in the source code ([5]). XPSL can specify constraints, aspects, queries for a source code, in which the results can be further fed into another software analysis like data race detection and code assurance analysis, making the code size to be analyzed significantly smaller than the entire code.

William Scherlis’s work on static analyses also looks into the possibility of utilizing users’ concerned aspects. Its source code model-checking program requires programmers to annotate source code to do code assurance ([12], [23], [15]). By parsing the annotations, the program can decide if a file of code should be analyzed. [16] looks into the possibility of providing a query-based system for users to specify constraints on source code annotation.

Sunsoft’s WARLOCK([22]) data race tool maintains a repository of data race information files (each file corresponds to a source code file, generated upon compilation). To do static data race checking, users directly specify which files to analyse, with additional input from user to improve the efficiency of the analysis.

In Context-Sensitive Control Flow Graphs, we require users to provide a set of program points in the source code to build the graphs on. With the help of these program points, we determine files and methods that are relevant to the points and only analyze those methods. Therefore we save the redundant work of analyzing irrelevant code.

### 2.3 Representing Concerns in Graphs

[17] works on representing user’s concerns in graphs, in which they call concern graphs. Their work finds and visualizes concerns using structural dependencies between procedures in Java. The concern graphs addresses relationships between procedures such
as read/write, declarations, superclass and caller/callee, depending on user's specifications.

In our Context-Sensitive CFG, we represent user's concerns using CFGs and call order graphs, in which we only shows part of the original CFGs and call order graphs that are relevant to the program points the user is interested in.
CHAPTER 3.  Context-Sensitive Control Flow Graph

In this chapter, we discuss the core idea of building Context-Sensitive CFG in details.

3.1 Complexity of Graphs

Figure 3.1 shows an ordinary CFG for the method ipproc in XINU operating system. As we can see, the graph is so huge that it cannot be seen clearly when fit within an A4 size paper. To reduce the size of this graph, we need a bearing. We defined the complexity of a graph to be directly addressed by the amount of nodes and edges the graph contains. In this thesis, our objective is to reduce the complexity of CFG by reducing the amount of nodes and edges displayed.

3.2 Marking

Programmers often have certain points of interest or aspects in mind when they try to understand a program. These points of interest can eventually boil down to statements in the program source code. For example, a programmer might be interested in how a mutex is being locked and unlocked in a program. In this case all method calls to lock and unlock a mutex in the program are the points of interest of the programmer. We refer to these points of interests as markings.
Figure 3.1  A sample CFG of method ipproc() in XINU operating system
Markings can be any program entities such as variable assignments and method calls. In building Context-Sensitive CFG, our goal is to make these markings visible on CFGs no matter how high level of our graph view is. These markings serve as our compass in dealing with huge size code and shrinking our CFGs. It is notable that if a set of markings has an aspect in common, this set of markings is a pointcut in Aspect-Oriented Programming ([1]).

3.3 Relevance of Program Entities

In building Context-Sensitive CFG, once we acquired the set of markings from user, we then have to determine program entities that are related to this set of markings. In CFG, a basic program entity is a statement. Hence we will have to check all statements in the program and determine if they are relevant to the set of markings specified. As mentioned earlier, if a statement is relevant, it will be shown in the resulting graph, else it will be hidden.

We break down Context-Sensitive CFG into intra-procedural level Context-Sensitive CFG (within a method) and inter-procedural level Context-Sensitive CFG (among methods). The approach of determining relevance of statements towards markings slightly differs for these 2 cases. Other than Context-Sensitive CFG, we also build Context-Sensitive Call Order Graph (COG) to help in navigating between methods. The basic program entity for a COG is a method, which is different from a CFG. The approach of determining relevance of methods towards markings is different from CFG too.

3.4 Intra-procedural Context-Sensitive CFG

In this section we first look into intra-procedural CFG. A CFG is a graph that shows the control flow in a program. Take for example a simple C code as follows:
if( a < 1 )
   a=2;
else
   a=0;

The resulting graph is shown in Figure 3.2.

An intra-procedural CFG is a graph that represents the control flow of a method using the approach that is just discussed. It starts with a method start node indicating the start of a method, and ends with a method end node. Each statement in the method is represented with at least a node.

An intra-procedural Context-Sensitive CFG tries to hide irrelevant information according to user-specified markings. As said in previous section, the basic entity for a CFG is a statement. To determine information that are relevant to markings is to determine what statements are relevant to the markings. We specify statements that are relevant to markings as all ancestral statements of the markings. In other words, all block statements that markings directly or indirectly nested in.

Consider the C source code as a tree-like structure, the relevant area of a marking which are directly of interest to the user are usually the parent blocks that the marking is nested in, because they best show the control flow that reaches the marking. Figure
3.3 illustrates what are the siblings of a statement and the parent and ancestor blocks of a statement. In the example demonstrated in Figure 3.3, the marking is the call made to method \texttt{bar()}. The code area that is not in grey is the area that are relevant to the marking. They will be represented with one node per statement. As for areas in grey, they will be collapsed into basic block nodes and they will be labeled with unique ID. A basic block node is a linear sequence of program statements with one entry and one exit point, a variation of definition by [8]. The resulted Intra-procedural Context-Sensitive CFG is shown beside the code.
3.5 Inter-procedural Context-Sensitive CFG

Before we look into inter-procedural Context-Sensitive CFG, it is necessary to look into a regular inter-procedural CFG first. Traditionally, building inter-procedural CFG involves concatenating a method's intra-procedural CFG to all its calling context. Although this approach results in a huge and incomprehensive graph, inter-procedural CFG was traditionally not meant to be visualized. The standard approach nowadays is to use a call reference node at each calling context that refers to the start node of the called method's intra-procedural CFG. This approach brought CFG closer to being visualize.

In building inter-procedural Context-Sensitive CFG, we use the same approach as previously mentioned. However, there is a need to discuss the relevance of statements (statements are still the basic entities) for inter-procedural Context-Sensitive CFG. Markings do affect statements in other methods inter-procedurally. If a call is made to a method that contains at least one marking, the calling statement is then considered to be a marking. Hence all control structure blocks that the calling statement directly or indirectly nested in will be retained in inter-procedural Context-Sensitive CFG. Figure 3.4 demonstrates an example Context-Sensitive CFG for a method that contains a call to another method that contains markings.

3.6 Context Sensitive Call Order Graph

Inter-procedural level analyses always involve call order analysis to resolve relationships among methods. Unlike other analyses like data flow and pointer analysis, the results of a method's control flow analysis remains unchanged across different context calls. This means that once a CFG of a method is computed, it is applicable to all calls made to this method. This remains true even for our Context-Sensitive CFG because the markings inside a method will not change with respect to context calls. Hence, to
build inter-procedural Context-Sensitive CFG, we only need an accurate call order tree that links all intra-procedural Context-Sensitive CFG together.

The call order graphs for large software systems are always big forests of trees that are almost impossible to comprehend. To shrink the size of the call order graph, we determine methods that are relevant to the markings specified (the basic entity of a call order graph is a method). By showing only methods that contains markings and other methods that are relevant to them, we can provide a good and comprehensible call order graph. The resulting call order tree, which we refer to as Context-Sensitive Call Order Graph, not only shows relevant information about user’s specified markings, it also reveals some underlying architecture of the software system that is buried within the code.

As an example, figure 3.5 shows a call order graph build with node “A” as root method. “M1” and “M2” are methods that contains markings. In building Context-Sensitive Call Order Graph, node “E” and “F” will be left out from the graph because they do not lead to a method that contains markings, and hence they are both “irrelevant” to the user-specified context. The implementation of the Context-Sensitive Call Order Graph will be further discussed in Chapter 4.
One issue that arises as we are developing Context Sensitive Call Order Graph is, which method should we start building the call order graph on. In other words, which method should be the root of the graph. In an attempt to solve this problem, we start from the markings and recursively follows the call order relation bottom up to figure out the root of the program. We refer to these methods as root methods. The disadvantage of doing this, is that it is highly possible that the graph might end up having multiple roots and fairly large to be comprehensible. It is also computationally intensive and hence this approach is discouraged. To resolve this issue, we let users decide which methods should be used as the root methods of the call order graphs.

3.6.1 Context-Sensitive Node Deletion

A Context-Sensitive Call Order Graph can still be too large to comprehend if the marking set that a user specified cross cuts too many methods. In this situation, users might decide that certain method nodes in the Context-Sensitive Call Order Graph might not be interesting to look at, even if those methods call other methods that contain markings. Users can then specify the methods to delete from the Context-Sensitive Call Order Graph.

A node deletion in Context-Sensitive Call Order Graph works differently from a
regular graph node deletion. All nodes that are only reachable from this node will be deleted. All nodes that have become “irrelevant” to the marking set due to this deletion will be deleted as well. Figure 3.6 demonstrates a Context-Sensitive node deletion. Nodes that contains markings are “M1”, “M2” and “M3”. When the node “D” is being deleted, node “E” and “M3” are being deleted even when “M3” contains markings, simply because node “M3” is no longer reachable from node “A”. Do note that node “C” is being deleted as well because it no longer leads to any node that contains a marking.
CHAPTER 4. ContextCFG tool and its Implementation

We programmed a tool called ContextCFG that builds Context-Sensitive CFG. It is built using Java programming language, and targets the C programming language. In this chapter, we will discuss in details how the implementation of ContextCFG is being carried out. The basic procedures of using ContextCFG tool is given at the end of the chapter.

4.1 Overview

Figure 4.1 shows the architecture of ContextCFG. Before the actual construction of the Context-Sensitive CFG can be carried out, some preprocessing work is done to ease the construction process. Then the graph construction module constructs all graphs needed for visualization and passes the result to visualization module.

4.2 Preprocess Work

The source code files are converted into eXtensible Intermediate Common Language (XCIL) files. All analyses in this thesis work on top of these XCIL files. The XCIL is XML tree-based representation of source code. We utilize the JDOM library, which is an open source library, for traversing the XCIL.
4.2.1 eXtensible Common Intermediate Language (XCIL)

The eXtensible Intermediate Common Language (XCIL) is a language independent common format that represents source code’s semantics ([5]). It is a good starting point for source code analysis. It was developed by Ensoft Corp and utilizes the EDG front end designed by Edison Design Group ([4]). Figure 4.2 shows the process of converting C code from the source file into corresponding XCIL file. Since all intermediate files and data (including XCIL) are all in XML format, we utilize the Java Data Object Model (JDOM) package for the ease of parsing these files ([2]).

4.2.1.1 EDG Front End

The EDG front end works like a compiler, except that instead of generating machine code for execution, it generates semantic information in XML format and stores it in
Figure 4.2 Conversion from C code to XCIL

`.c file` → `EDG Front End` → `XCIL Filter` → `XCIL file` → `xcil.xml` → `callorder.xml` → `Call Order Analysis`

*edg.xml*. It is capable of parsing C and C++.

### 4.2.1.2 XCIL filter

The output of the EDG front end is a big XML file that is hard to parse. The XCIL filter takes this XML file as input and outputs to `xcil.xml`. The result file is considerably easier to parse as compared to *edg.xml* because it conforms to a more comprehensible format. Figure 4.3 shows a sample snippet of C code converted into XCIL data. One important property of the XCIL is that the attribute “id” for each XCIL element are always unique even across multiple files. This property is very essential in doing interprocedural analysis because we need to uniquely identify methods and global variables.

### 4.2.2 Call Order Analysis

The call order analysis extracts information from the XCIL files of the source code and stores it in an xml file (*callorder.xml*). It extracts all methods in the source code into a list, with all method calls information nested within the corresponding method. Figure 4.4 shows a view of how *callorder.xml* is structured. Function pointer calls are currently not handled but it is a good and important future work to be completed.
A source code program point searching tool is indeed needed to helping specifying markings in code. This tool is very important to address user's points of interest. The more specific the marking set is, the more comprehensible our Context Sensitive CFG will be.

Currently, markings are specified by users via our ContextCFG tool, and it currently only allow users to specify method calls as markings. Our ContextCFG tool searches for all XCIL references corresponding to the method calls specified. The corresponding XCIL ids of the markings and their corresponding files names and paths are kept in a list. This markings list will be dumped into an XML file named by user when user is done with specifying markings. Figure 4.5 shows how a marking is acquired from source code, and converted into XML format for later use.

It is also possible to have another tool that does point searching to feed in our graph.
- `<Method name="addarg" visibility="public" id="54" storageClass="unspecified" 
  compilerGenerated="true" filename="./shell/addarg.c.xbundle/xcil.linked.xml"> 
  <GlobalMethodCall id="288" operation="578" type="430" storageClass="" methodID="117743" 
    method="strcpy"/>
  <GlobalMethodCall id="171" operation="282" type="430" storageClass="" methodID="117835" 
    method="strlen"/>
</Method>

- `<Method name="ascdate" visibility="public" id="995" storageClass="unspecified" 
  compilerGenerated="true" filename="./shell/ascdate.c.xbundle/xcil.linked.xml"> 
  <GlobalMethodCall id="1247" operation="1031" type="1033" storageClass="" methodID="116465" method="sprintf" />
</Method>

Figure 4.4 The structure of `callorder.xml`

```
if (optr->pstate == PRCURR) {
    optr->pstate = PRREADY;
    insert(currpid, readyhead, optr->pprio);

    CXIL

    C code
    MARKING
    file
```

Figure 4.5 Generation of markings XML file

construction module. The eXtensible Program Specification Language (XPSL) of the 
KCS framework might be well suited for this task ([6]).

### 4.4 Graph Construction Module

The graph construction module can be divided into 2 parts: construction of Context-
Sensitive Call Order Graph and Context-Sensitive CFG at intra-procedural level. Note 
that we didn’t include the construction of inter-procedural Context-Sensitive CFG be-
cause it is being addressed by combining Context-Sensitive Call Order graph and intra-
procedural Context-Sensitive CFG, as mentioned earlier. However, we will discuss about building intra-procedural Context-Sensitive CFG for methods that does not directly contain markings later in this chapter.

### 4.4.1 Building Context-Sensitive Call Order Graph

The construction of Context Sensitive Call Order Graph requires 3 inputs: markings file, call order file, and the user specified root method for the graph. The Context Sensitive Call Order Graph is done in a 3-pass process. The first pass constructs and prunes the call order tree of the software system starting from the root method. Second pass marks up the methods that contain markings in the newly build graph, then the third pass prunes of irrelevant call nodes in the graph according to the marks from second pass. The following subsections describe each pass in details

#### 4.4.1.1 Call Order Graph Construction

To build the call order graph, we read in the call order file (`callorder.xml`) and stores the information in a JDOM tree. We then recursively traverses the JDOM tree starting from the user specified root method, building and linking nodes together along the way. The recursive function stops recursing if the current method has no calls, or if the method node for current method is already created. The result graph will be a call order graph reachable from the root method. Figure 4.6 gives a look on how the recursive function is structured. The data structure to store a call graph node is fairly simple (Figure 4.7).

#### 4.4.1.2 Marking Up Call Order Branches

Once we have the markings from user, we compute a list of methods in the system that contains these markings. We then determine the intersection set between this list of methods and the set of methods in the previously built call order graph. Starting from each method in the intersection set, we recursively traverse the call order graph
public CallOrderGraph graph;
public void COGTopDown(Element COTmethod, CallOrderGraphNode prev){
    CallOrderGraphNode aNode;
    if(!traversedCalls.contains(COTmethod)){ // method is not previously traversed
        aNode = new CallOrderGraphNode(COTMethod);
        prev.callee.add(aNode);
        aNode.caller.add(prev);
        graph.add(aNode);
    } else{ // method already traversed
        CallOrderGraphNode aNode = graph.getNode(COTmethod);
        prev.callee.add(aNode);
        aNode.caller.add(prev);
    }
    if(COTmethod.hascallee){
        Iterator i = COTmethod.calleeset.iterator();
        while(i.hasNext())
            COGTopDown((Element)i.next(), aNode);
    }
}

Figure 4.6 Recursive function traverses callorder.xml's JDOM tree to create Call Order Graph

bottom up and set the relevant flag of every node we encounter along the way (refer to Figure 4.7). The relevant flag simply indicates whether this method is relevant to the user's marking set.

The reason why we do not prune the call order graph in the second pass is because the graph traversal algorithm is recursive. Imagine that the program currently reaches a node in the call order graph. It cannot make decision whether its callee can be pruned off because its callee might lead to another method that has markings, but it would not know because that part of the graph is not traversed yet. Hence, the work of pruning the tree has to be pushed to the third pass.

public class CallOrderTreeNode{
    public boolean hasMarking = false;
    public boolean relevant = false;
    private String name;
    private Identifier ID;
    private Element xclref;
    private ArrayList caller; //Type: CallOrderTreeNode
    private ArrayList callee; //Type: CallOrderTreeNode
}

Figure 4.7 The Call Order Graph Node class and its fields
4.4.1.3 Pruning the Call Order Graph

The third pass is a very straightforward process. Starting from the root method again, we traverse down the graph, unlinking nodes that do not have their relevant flag set to true.

4.4.2 Building Intra-procedural Context-Sensitive CFG

We build the Context Sensitive CFG for a method only when user requests to view it. The Context Sensitive Call Order Graph gives user a good sense of what methods that he might be interested to look into. Upon request, our tool will then generate a Context Sensitive CFG for the method.

Context Sensitive Control Flow Graph requires 3 inputs: markings file, a user-specified method and the XCIL file that corresponds to the method's source file. The process of generating Context Sensitive CFG happens to be simpler as compared to Context Sensitive Call Order Graph. It only requires 2-pass graph traversals. However, the data structure for its graph node is more complex. Figure 4.8 shows the data structure of the Context Sensitive CFG node. In following subsections, we will discuss each pass in details.

4.4.2.1 Determining Relevant Nodes

The first step of building the Context Sensitive CFG is to determine the relevant nodes of a marking. Once the XCIL file of the method is being loaded into JDOM tree, we searches the tree for all the markings in the method, and start traversing the JDOM tree bottom up to collect all markings' ancestor' XCIL ids into a set. This set is needed in graph construction describe in next subsection (Refer to Figure 4.3 for XCIL format structure)
public class CFGraphNode {
    public Identifier ID;
    public String label;
    public int type; // Node type; if-else, while, etc.
    public boolean isMark = false;
    public boolean relevant = false;
    public boolean inPath = false;
    public ArrayList PathNext;
    public boolean display = false;
    
    // Next nodes if this node is collapsed (not relevant to markings)
    public ArrayList collapsedNext;
    // From nodes if prev node is collapsed
    public ArrayList collapsedFrom;
    
    // Next nodes if this node is expanded (relevant to markings)
    public ArrayList expandedNext;
    // From nodes if prev node is expanded
    public ArrayList expandedFrom;
    
    // End node for this node; i.e.: dummy End node for switch, while, do-while, for etc.
    public CFGraphNode endNode;
    
    // XCIL reference of this node
    public Element xcilRef;
    ...
    // constructors and methods
}

Figure 4.8 The Context Sensitive CFG Node class and its fields

4.4.2.2 Building the Graph

The Context-Sensitive CFG construction starts from the XCIL element for the method in JDOM tree and recursively traverse down the tree in top down fashion. Appropriate graph nodes are build and linked along the way. If a node's XCIL ID happens to be in the previously computed relevant ID set, then the node's relevant flag will be set to true. Unlike the Context Sensitive Call Order Graph, here we do not prune the graph because all other irrelevant nodes are to be collapsed into basic block nodes in later presentation. To provide the feature of expanding basic block nodes, these irrelevant nodes can not be pruned away. (Refer to Appendix 1 for the actual Java code for implementing the Context Sensitive Control Flow Graph)

4.4.2.3 Final presentation

The final presentation of the Context-Sensitive CFG is done by the visualization module. This module simply traverses the graph and build nodes along the way. If a node is irrelevant, this module decides to either build a new basic block node or
concatenate this node to an already existing basic block node. If a node is relevant, a node representing it is built.

4.4.3 Building Inter-procedural Context-Sensitive CFG

Once the program for building intra-procedural Context-Sensitive CFG is done, building inter-procedural Context-Sensitive CFG is a lot easier. Recall from previous chapter where we discussed about relevance of statements in inter-procedural Context-Sensitive CFG. All calls made to a method that contains a marking is treated as markings as well. Hence, from the call order analysis, we find methods that are relevant to the user-specified context and run these methods through the process of building intra-procedural Context-Sensitive CFG.

4.5 Visualization module

We use the standard Java Swing class to implement our graphical user interface (GUI). For graph drawing, we utilized the AT&T open source graphics library called Grappa ([7]). We also utilizes an open source graph layout manager program called Dotty to provide good layout for the graph([3]). All graphs instances passed in from the graph construction module are converted into Grappa graph structure before visualizing them in the GUI. The conversion for Context Sensitive Call Order Graph is a straight forward node-to-node conversion. However, the conversion for the intraprocedural Context Sensitive CFG requires more work.

As mentioned earlier, Context Sensitive Control Flow Graph represents nodes that are irrelevant to markings in basic block nodes. The basic block node representation is implemented only at visualization level. As the visualization program traverses the Context Sensitive CFG, it builds a Grappa node for every node that has its “relevant” flag set to true. When the visualization program encounters a node that does not have
the flag set to true, program will build a basic block node and collects irrelevant nodes into the basic block node. The program will stop collecting irrelevant nodes when it encounters another relevant node, which requires it to build a new Grappa node, or the end of a control structure block.

Whenever user double-clicks a basic block node, the program will determine all nodes that are collected within the basic block node, set them to relevant and call the graph construction module to reconstruct the Context Sensitive CFG with added relevant nodes.

4.5.1 Procedure on using ContextCFG tool

Figure 4.9 demonstrates a screenshot of the ContextCFG tool. To use the tool, first the user has to select the piece of software that he wants to analyze. In this step, we
only require user to select the parent directory where all the source code files reside. The ContextCFG will recurse into all subdirectories to analyze all source code files. The tool will then run all the preprocessing work on the software (generating XCIL and call order file). This process will take some time. Once it is done, there is no need to rerun this process unless user wish to analyze a different piece of software.

User then has to select a marking file, or specify one if he has none. Once the marking file is selected, a window will pop up showing a list of all methods that contain the markings. Double clicking on any method in the list will show the intra-procedural Context-Sensitive CFG for that method.

To generate the Context-Sensitive Call Order Graph, user has to select the option from the menu and specify the root method. The graph will be displayed once the root method is provided. As a feature, user is also able to context-sensitively delete a node from the graph to minimize the size of the graph (refer to Chapter 3.6.1). Similarly, double clicking on any node in a Context-Sensitive Call Order Graph will prompt the tool to display the intra-procedural Context-Sensitive CFG for that method node.
CHAPTER 5. Results

We test ran our program on the XINU operating system codebase. The case study that we did on XINU operating system code shows satisfactory results.

We did a case study on the memory allocation aspect in XINU in which we study the control flow of `getbuf()` and `freebuf()` calls throughout the operating system codebase. Hence we include all `getbuf()` and `freebuf()` calls into our marking set.

5.1 Graphical Results

5.1.1 Intra-procedural level

In this section we will compare an ordinary intra-procedural CFG with an intra-procedural Context-Sensitive CFG. We first look at an ordinary call order graph for `ipproc()` method (Figure 5.1). As we can see, with proper scaling, the graph is hardly visible in a page. Compared this CFG with the intra-procedural Context-Sensitive CFG for `ipproc()` with `getbuf()` and `freebuf()` as markings (Figure 5.2).

5.1.2 Inter-procedural level

Again, in this section we compare an ordinary inter-procedural CFG with an inter-procedural Context-Sensitive CFG. Because both graphs are build just by hyperlinking intra-procedural CFGs together, there aren’t any actual graphs to show. However, comparing ordinary Call Order Graph and Context-Sensitive Call Order Graph gives
Figure 5.1 Ordinary CFG for ipproc()
Figure 5.2  Context-Sensitive CFG for ipproc() with getbuf() and freebuf() as markings
a good high level view of how is inter-procedural CFGs compared to inter-procedural Context-Sensitive CFGs.

In Figure 5.3, we demonstrate the ordinary Call Order Graph with \texttt{iproc()} as a root method of the call order graph. Again, the call order graph is so huge that visualizing them on paper shows nothing more than a blur. Figure 5.4 show the Context-Sensitive Call Order Graph with \texttt{iproc()} as root method, and the marking set being the same as previous section.

As we can see, the complexity of Call Order Graph is highly reduced in Context-Sensitive Call Order Graph. However the Context-Sensitive Call Order Graph is still fairly large for comprehension. Suppose we want to shrink the graph more by deleting some nodes. A domain expert in XINU would find that \texttt{panic()} method should be pruned out of the graph because it is a starting point for debugging harness, and it creates a huge loop in the call order graph when it is not really of interest. So next we will have a look at the same Call Order Graph but with \texttt{panic()} pruned off (Figure 5.5). The graph still shows nothing better, even though the graph size has been significantly reduced. Figure 5.6 shows the same Context-Sensitive Call Order Graph, but this time with \texttt{panic()} method deleted context-sensitively (refer to Chapter 3.3.1 about context-sensitive node
Figure 5.4 Context-Sensitive Call Order Graph with ipproc() as root method, and getbuf() and freebuf() as markings.
Now we see a good and clear picture of Context-Sensitive Call Order Graph that will take a programmer a few days of work to figure out how the transition of control between methods happens.

5.2 Quantitative Results

To show quantitative results, we utilize the same markings set used in previous section (all `getbuf()` and `freebuf()` calls). In this section we will show 2 tables comparing the complexity of our Context-Sensitive CFG and Call Order Graph as oppose to ordinary CFG and Call Order Graph.

Table 5.7 shows total nodes and edges count of a normal CFG compared to Context Sensitive CFG. The 5 methods represented in the table are randomly selected from the method set in XINU operating system that contains `getbuf()` or `freebuf()` calls.

As we can see some methods achieve higher reduction while others lower. The reason this is happening is because the reduction percentage is directly affected by the amount of markings that a method contains. Some methods have more markings and hence, and lower reduction percentage is achieved.
Figure 5.6 Context-Sensitive Call Order Graph with ipproc() as root method and panic() deleted context-sensitively
<table>
<thead>
<tr>
<th>Method</th>
<th>Ordinary</th>
<th></th>
<th>Context-Sensitive</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nodes</td>
<td>Edges</td>
<td></td>
<td>Nodes</td>
</tr>
<tr>
<td>icmp</td>
<td>92</td>
<td>97</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>dgwrite</td>
<td>76</td>
<td>80</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>tcpinp</td>
<td>97</td>
<td>102</td>
<td>24</td>
<td>26</td>
</tr>
<tr>
<td>ipproc</td>
<td>179</td>
<td>192</td>
<td>46</td>
<td>53</td>
</tr>
<tr>
<td>netwrite</td>
<td>128</td>
<td>135</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Figure 5.7 Total nodes and edges count comparison between ordinary CFG and Context-Sensitive CFG

\[
\text{reduction} = \frac{\text{Context-Sensitive graph's count}}{\text{Ordinary graph's count}} \times 100\%
\]

Figure 5.8 Formula for calculating reduction percentage

To calculate the average reduction percentage, we build Context-Sensitive CFGs for all methods that contain markings in XINU operating system and record down the nodes and edges count for each of them. For each method, we compare its Context-Sensitive CFG's nodes and edges count with its ordinary CFG's by dividing the Context-Sensitive graph's count with the ordinary graph's count. We then multiply the results with 100 to obtain the reduction percentage. Figure 5.8 shows formula for calculating reduction percentage.

Once we have the reduction percentage for all methods, we can calculate the average, min and max for this set of methods. In this particular case study, in which we uses all `getbuf()` and `freebuf()` calls as markings in XINU operating system, we found that the average node reduction is 81.1%, with maximum and minimum reduction at 96.6% and 59.1% respectively. The average edge reduction is 81.6% with maximum and minimum reduction at 97.4% and 59.1% respectively.

For comparing call order graph, we extracted a set of methods that are the root methods in XINU operating system (refer to chapter 3.6 for definition of root method). Using the same set of markings as before, we build call order graphs for each method.
<table>
<thead>
<tr>
<th>Method</th>
<th>Ordinary</th>
<th></th>
<th>Context-Sensitive</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nodes</td>
<td>Edges</td>
<td>Nodes</td>
<td>Edges</td>
</tr>
<tr>
<td>ipproc</td>
<td>143</td>
<td>628</td>
<td>62</td>
<td>233</td>
</tr>
<tr>
<td>dgwrite</td>
<td>88</td>
<td>344</td>
<td>27</td>
<td>54</td>
</tr>
<tr>
<td>main</td>
<td>110</td>
<td>451</td>
<td>34</td>
<td>74</td>
</tr>
</tbody>
</table>

Figure 5.9  Total nodes and edges count comparison between ordinary Call Order Graph and Context-Sensitive Call Order Graph

in this extracted set. Table 5.9 shows the graph size of a traditional call order graph compared to Context Sensitive call order graph.

Similarly, we calculated the average reduction percentage for call order graphs for this method set using the same formula as before. We found out that the average node reduction for call order graph is 69.2%, with maximum and minimum reduction at 78.3% and 53.2% respectively. The average edge reduction is 83.4% with maximum and minimum at 91.2% and 62.9% respectively.

As we can see from the quantitative results, the idea of Context-Sensitive CFG and call order graph has indeed reduces the overall complexity of ordinary CFG and call order graph and provides more comprehensibility. We manage to shrink down both types of graphs without losing information (information are only being hidden out, it can be recalled if needed). The quantitative result shows that this idea is indeed favorable.
CHAPTER 6. Program Path Searching Queries

When understanding programs, users often come across situations where they are not only interested in certain markings in the code, they also need to learn about the relationship between markings. For example, programmers often like to know where is a mutex is unlocked in the program after it is locked, or where a memory is being freed after it is allocated. Scenarios like this motivated us to further provide program path searching queries on top of our Context-Sensitive CFG. The ability of XCIL to address any specific program point in a program gives us the ability to specify a specific flow in a program, hence making program path searching queries possible to implement.

This thesis does not intend to implement a complete set of queries that solves all issues about program path searching queries. We have decided to provide queries that only operates within the scope of a method (intra-procedural). This means that the starting point and the ending point of a query should be within the same method. We formulated 2 kinds of path searching queries for path searching within a method:

1. Starting from one marking, determine program paths that lead to a specified set of end markings.

2. Starting from one marking, determine program paths that reach the end of method without encountering any markings in a specified set of end markings.

The first query, as mentioned before, can be used to determine paths between 2 markings like mutex lock and unlock. The second query can provide comprehension on problems like memory leaks, where memory was allocated but wasn’t freed locally.
6.1 Implementing Program Path Searching Queries

Program path searching queries are fairly simple to implement once the Context Sensitive CFG for a method is done, with the assumption that both the starting point and ending point are in the same method. They only require traversal of the built graph in certain fashion and “mark” up of the graph nodes traversed accordingly. The node structure for a CFG graph node is extended with a new field named “inPath” which is just a boolean flag. This field indicates if a node is in the flow that is part of the answer to a query.

In the following subsections we are going to discuss the implementation of the queries.

6.1.1 Program Path Search from a start marking to a set of end markings

The query recursively traverses the Context Sensitive CFG, and set the “inPath” flag of the node to true on every nodes we encountered starting from the start marking. Once an end marking is found, it will set the flag for that marking node, and stops the traversal down its path.

6.1.2 Program Path Search from a start marking to end of method without encountering a set of end markings

Similarly the query starts from the start marking and recursively traverses the graph. It sets the “inPath” flag for every node it encounters. However, if an end marking is reached, the query will stop and start backtracking to previous branch point, typically control structure nodes, setting the flag back to false on the way. The query then continues down the other path of the branch point. If all paths of a branch point encounters end markings, the backtracking will continue up the graph. In the worst case, the query will terminate at the start marking, in which no path was found.
6.2 Program Path Searching Queries Results

In this section we demonstrate the result graph of the 3 program path searching queries. We are interested in scenarios where the relation between markings are of interest. We will take an example in XINU operating system, where program concurrency aspect is of concern to the user. To simulate the queries, we take the `enq()` of XINU as example, where the `wait()` is the start marking, with the end markings set being all calls to `signal()`. Fig 6.1 shows the sample code that we ran our query on. The result graphs for the 2 queries are being shown in Figure 6.2 and Figure 6.3 respectively.

6.3 Future work on Program Path Searching Queries

Exploring the possibilities of program path searching queries is another worthy field of research. However, before diving into more flavors of queries, allowing larger scope on existing queries post good challenge in research work. The polymorphism concept in object-oriented programming language and function pointers is an important issue to
Figure 6.2 Query 1
Figure 6.3 Query 2
solve if the query scope is to go towards inter-procedural.

6.4 Related Work

The query approach is a widely used technique in software engineering. However, no work has been done on querying a program’s control flow up until now. Querying program flow on a control flow graph is actually extracting nodes and edges of the graph specified by the query. A closer work that has been done would be GOQL, which is a generic graph querying language (\cite{20}). Due to its generality, GOQL queries are geared towards treating graphs like database, with no flow-specific or path-specific queries available for use. We want our query to be able to walk CFG and return program path as the answer of the query along the way.
Bibliography


    http://dirac.ece.iastate.edu/sec/XCIL.htm.

    http://dirac.ece.iastate.edu/sec/XPSL.htm.


[9] Larry A. Barowski and James H. Cross II. Extraction and Use of Class Dependency
    Information for Java. In Proceedings of the 9th Working Conference on Reverse
    Engineering (WCRE’02), 2002.


