Event views and graph reductions for understanding system level C code

by

Srinivas Neginhal

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 Kothari, Major Professor
Manimaran Govindarasu
Samik Basu
Rana Biswas

Iowa State University
Ames, Iowa
2006

Copyright © Srinivas Neginhal, 2006. All rights reserved.
Graduate College
Iowa State University

This is to certify that the master's thesis of

Srinivas Neginhal

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
Dedication

This thesis is dedicated to my parents and my brother.
TABLE OF CONTENTS

LIST OF TABLES .................................................. vi
LIST OF FIGURES ................................................ vii
ABSTRACT ......................................................... viii
CHAPTER 1. Introduction ......................................... 1
CHAPTER 2. Event Views and Graph Reductions .................... 4
  2.1 Event Views ............................................... 4
  2.2 Graph Reductions ......................................... 7
     2.2.1 Control Flow Based Reduction ....................... 8
     2.2.2 Function Based Reduction ............................ 9
     2.2.3 Code Layout Based Reduction ....................... 11
     2.2.4 Event Based Reduction .............................. 11
CHAPTER 3. CVision Architecture ................................ 13
  3.1 The Backend .............................................. 14
  3.2 The Information Database ................................ 15
     3.2.1 File Table ........................................... 15
     3.2.2 Function Table ....................................... 15
     3.2.3 Block Table .......................................... 15
     3.2.4 Control Flow Table ................................... 16
     3.2.5 Statement Table ....................................... 17
     3.2.6 Function Call Table ................................... 17
     3.2.7 Variable Table ....................................... 17
3.2.8 Type Table .................................................. 18
3.2.9 Type Association Table .................................. 18
3.2.10 Read-Write Table ...................................... 19
3.2.11 Parameter Table ........................................ 19
3.2.12 Function Summary Table .............................. 19
3.2.13 Correlation Table ........................................ 21
3.2.14 Function Root Table .................................... 21
3.2.15 Include Hierarchy Table ............................... 21

3.3 The Client .................................................. 22
3.3.1 Function Views .......................................... 23
3.3.2 Variable and Type Views ............................... 24

3.4 Evolution and Performance .............................. 25
3.4.1 Effectiveness ............................................. 25
3.4.2 Performance .............................................. 27

CHAPTER 4. Case Studies ............................... 32
4.1 A Xinu Case Study .......................................... 33
4.2 A Linux Case Study ......................................... 36

CHAPTER 5. Related Work ............................... 41

CHAPTER 6. Conclusions ................................. 43

BIBLIOGRAPHY ................................................. 44

ACKNOWLEDGEMENTS ......................................... 47
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Size of some commercially available operating systems</td>
<td>1</td>
</tr>
<tr>
<td>2.1</td>
<td>Types of events supported by CVision</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>File table</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Function table</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>Block table</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>Control flow table</td>
<td>16</td>
</tr>
<tr>
<td>3.5</td>
<td>Statement table</td>
<td>17</td>
</tr>
<tr>
<td>3.6</td>
<td>Function call table</td>
<td>18</td>
</tr>
<tr>
<td>3.7</td>
<td>Variable table</td>
<td>18</td>
</tr>
<tr>
<td>3.8</td>
<td>Type table</td>
<td>18</td>
</tr>
<tr>
<td>3.9</td>
<td>Type association table</td>
<td>19</td>
</tr>
<tr>
<td>3.10</td>
<td>Read-Write table</td>
<td>19</td>
</tr>
<tr>
<td>3.11</td>
<td>Parameter table</td>
<td>20</td>
</tr>
<tr>
<td>3.12</td>
<td>Function summary table</td>
<td>20</td>
</tr>
<tr>
<td>3.13</td>
<td>Correlation table</td>
<td>21</td>
</tr>
<tr>
<td>3.14</td>
<td>Function root table</td>
<td>21</td>
</tr>
<tr>
<td>3.15</td>
<td>Include hierarchy table</td>
<td>22</td>
</tr>
<tr>
<td>3.16</td>
<td>Primary keys and indices</td>
<td>31</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>An example function</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Function <code>deliver_skb()</code></td>
<td>6</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Function <code>_netif_rx.schedule()</code></td>
<td>7</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>(a) Call graph of <code>dswrite()</code> (b) Reduced call graph of <code>dswrite()</code></td>
<td>9</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Code for function <code>netin()</code> - Network input daemon in Xinu</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>Code for function <code>dswrite()</code> - Disk write function in Xinu</td>
<td>12</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Architecture of CVision</td>
<td>13</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>User interface for function views</td>
<td>23</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>User interface for graphical representations of the code</td>
<td>25</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>User interface for graph reductions</td>
<td>26</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>User interface for variable view</td>
<td>27</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Sample code</td>
<td>28</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>User interface for type view</td>
<td>29</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Code for function <code>dskenq()</code></td>
<td>35</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Code for function <code>dskqopt()</code></td>
<td>36</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Code for function <code>dsinter()</code></td>
<td>37</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Reduced call graph of <code>e1000_intr()</code></td>
<td>38</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Diagram depicting the use of aggregate data structures to link two</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>disjoint threads of execution</td>
<td></td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Function <code>net_rx.action()</code></td>
<td>40</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>Reduced call graph of <code>net_rx.action()</code></td>
<td>40</td>
</tr>
</tbody>
</table>
ABSTRACT

Concurrent processing, runtime bindings and an extensive use of aggregate data structures make system level C codes difficult to understand. We propose event views and graph reductions as techniques to facilitate program comprehension. Starting with some domain knowledge, a user can apply these techniques to quickly identify and analyze exactly those parts of the program that are relevant to a given concern. We have built a tool called CVision to demonstrate applicability of the proposed techniques. CVision is an interactive tool that allows the user to: (a) quickly get to the relevant parts of the code, (b) graphically visualize relationships between program elements, (c) interactively apply different graph reductions to eliminate irrelevant relationships. Using these capabilities, the user can quickly distill a large body of code and extract meaningful views of runtime events that capture the user’s concern. The proposed program comprehension techniques are demonstrated through two case studies based on Linux and XINU operating systems.
CHAPTER 1. Introduction

Program Comprehension is an important part of almost all software maintenance activities. Large software evolves over decades, making sustenance the most expensive and prolonged activity [5, 21, 3]. It is estimated that engineers spend nearly 90% of the time in program comprehension as a part of various sustenance tasks [3, 13].

C continues to be the programming language of choice for system level development. It is extensively used in operating systems and embedded systems. Over the decades, a huge code base has been developed in C. Table 1.1 shows the size of some of the commercially available operating systems [1]. Approximately 80% of this code is written in C [22]. Its maintenance entails understanding highly complex code with many intricate artifacts.

Table 1.1 Size of some commercially available operating systems

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Size in Source Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 2000</td>
<td>29 Million</td>
</tr>
<tr>
<td>Linux 2.6.0</td>
<td>6 Million</td>
</tr>
<tr>
<td>Sun Solaris</td>
<td>7.5 Million</td>
</tr>
</tbody>
</table>

Concurrent processing, runtime bindings and extensive use of aggregate data structures like queues, linked lists etc are the hallmarks of system level programs. As an example, consider the problem of deciphering how a typical network protocol stack is designed and implemented in an operating system kernel. One starts with some high-level domain knowledge summarized as follows: A network device interrupts the processor upon receiving a packet. The device interrupt handler receives the packet and puts it into a receive queue. Subsequently, in a different thread of execution, the kernel dequeues the packet and processes it. Based on
the packet type, the kernel directs the packet to the appropriate protocol handler. Runtime bindings, achieved through function pointers, provide for efficient and modular implementation of the protocol stack.

Given a particular operating system kernel, how do we bridge the gap between our high-level domain knowledge and a specific implementation of the protocol stack? In particular how do we answer questions such as:

**What are the different C functions involved in routing a packet through different layers of the protocol stack? What are the places in the code where memory for the packet is allocated and later deallocated? What are the shared aggregate data structures through which a packet gets communicated between different kernel threads? What are the synchronization mechanisms and how are they applied to protect consistency of the shared aggregate data structures? What are the various runtime bindings and how are they applied to achieve a modular design?**

As mentioned earlier, we assume some high-level domain knowledge as the starting point for program comprehension. The objective of our approach is to provide an interactive tool to:

(a) allow the user to apply such domain knowledge, (b) quickly extract the relevant and exact details of the program, (c) provide an interactive environment to explore different possibilities that an user can construe based on the program details revealed by the tool. To achieve the objective, we have introduced event views and graph reductions techniques and built an interactive tool called CVision which implements these techniques.

Domain knowledge as a starting point is essential for program comprehension. For example, if the user has no knowledge about semaphores, it is impossible for the user to understand programs that use semaphores. The purpose of a tool such as CVision is to assist users in bridging gaps between their domain knowledge and specific details related to its implementation. For instance, the tool can help users uncover the specific details of how semaphores are used in a given program and determine if there is a possibility of a deadlock between different threads of execution. Maintenance engineers have the domain knowledge. Mapping this domain knowledge to specific implementation details in a large and complex code base is difficult and time consuming. Thus, it is valuable to have a tool that enables the users to discover code
using domain knowledge.

The thesis is organized as follows. Chapter 2 describes the code comprehension techniques we use. We introduce *event views* in Section 2.1. In Section 2.2 we discuss interactive techniques to manage complexity in graphical representations of large code. We present the architecture of CVision in Chapter 3. In Chapter 4, we present two case studies to demonstrate the capabilities and effectiveness of CVision. In Section 4.1 we analyze the Xinu operating system code to study a possible memory leak. In Section 4.2 we present a case study based on Linux TCP/IP protocol code. Chapter 5 compares CVision with the other program comprehension tools.
CHAPTER 2. Event Views and Graph Reductions

Developers use top-down, bottom-up or mixed approach to create a mental picture of the code [11]. A code comprehension tool should provide the user with micro and macro level views of code to help him build this mental picture. Micro level views consist of the implementation details. The developer usually has to isolate and comprehend minute details in the code to develop a micro level understanding of the code. Macro level views consist of a coarse, overall picture of the program behavior. Graphical representations like call graphs, control flow graphs etc help the developer acquire a macro level understanding of the code. In this chapter, we introduce two techniques used in CVision. Event Views help the developer gain a micro level understanding of the code by allowing the user to define and query for interesting behavior in the code. Graph Reductions allow the user to extract meaningful graphical representations of the code which help in macro level understanding of the code.

2.1 Event Views

When using a bottom-up approach, developers start off by looking for interesting events manifested in the code and gradually abstract out the details. The events of interest vary depending on the developer’s concern. For example, when a developer is trying to isolate memory leaks in the code, the events of interest would be memory allocations and deallocations, aliasing of pointers and pointer escapes through function calls. CVision enables users to isolate and navigate through the code based on such events. It helps users to construct views of relevant parts of the program that focus on a specified set of events. We refer to these as event views. In this chapter, we discuss a variety of events and illustrate how event views can help users in understanding large system level C code.
CVision recognizes a semantically rich set of program artifacts such as statements, functions, variables, types etc. In addition, CVision recognizes properties associated with these artifacts. For example, every variable in the code has a scope associated with it, hence scope is a property of a variable. Further, the program artifacts can be inter-related. For example, a statement \( sI \) could belongs to a function \( fI \) or a variable \( vI \) could be of type \( tI \). Events in the code are defined with respect to these program artifacts, their properties and their inter-relationships. We define an event as an operation on a program artifact. For example, read of a variable is an event. CVision allows the user to form complex event definitions and isolate such events in the source code. For example, the user could query for all the places where a global variable \( vI \) is being read in a test condition in a function \( fI \). In the rest of this section we provide examples of how event views help in isolating and understanding complex program behavior.

Table 2.1 gives an example of events in CVision. The table uses the code in Figure 2.1.

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Program Artifact</th>
<th>Identifier</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Variable</td>
<td>dev</td>
<td>Write of a Variable</td>
</tr>
<tr>
<td>4</td>
<td>Type</td>
<td>struct.net_device</td>
<td>Write of a Type</td>
</tr>
<tr>
<td>4</td>
<td>Function</td>
<td>kmalloc</td>
<td>Call to kmalloc()</td>
</tr>
<tr>
<td>5</td>
<td>Variable</td>
<td>dev</td>
<td>Read of a Variable</td>
</tr>
<tr>
<td>5</td>
<td>Variable</td>
<td>dev</td>
<td>Use of a Variable in a Condition</td>
</tr>
<tr>
<td>5</td>
<td>Type</td>
<td>struct.net_device</td>
<td>Read of a Type</td>
</tr>
<tr>
<td>5</td>
<td>Type</td>
<td>struct.net_device</td>
<td>Use of a Type in a Condition</td>
</tr>
<tr>
<td>8</td>
<td>Variable</td>
<td>dev</td>
<td>Passed as a parameter to kfree()</td>
</tr>
<tr>
<td>8</td>
<td>Type</td>
<td>struct.net_device</td>
<td>Passed as a parameter to kfree()</td>
</tr>
</tbody>
</table>

1. int foo()
2. {
3. struct net_device *dev;
4.   dev = kmalloc(sizeof(struct net_device), GFP_KERNEL);
5.   if (NULL == dev) {
6.     return -1;
7.   }
8.   kfree(dev);
9. }  

Figure 2.1 An example function
Based on domain knowledge, the developer can use these events as queries or to compose event views which unravel the intricacies of the system. For example, by looking for events where kswapd gets passed as a parameter to the function kernel_thread(), a Linux developer can quickly find the launching places for kswapd kernel thread. Note that besides the general notion of kernel threads which is common domain knowledge, the user needs to specifically know that kernel threads are launched using the function kernel_thread() in Linux.

It is common for developers to search for specific uses or modifications of certain program artifacts. For example, consider the code for deliver_skb() shown in Figure 2.2. deliver_skb() is a function in the Linux TCP/IP source code. This function is a typical example of the runtime binding in Linux. A developer trying to understand this code would be interested in knowing all the functions that the function pointer func binds to. CVision helps user navigate to places in the code where the function pointer func is modified or initialized so the developers can quickly resolve the run-time binding.

```c
static __inline__ int deliver_skb(struct sk_buff *skb, struct packet_type *pt_prev)
{
    atomic_inc(&skb->users);
    return pt_prev->func(skb, skb->dev, pt_prev);
}
```

Figure 2.2 Function deliver_skb()

As mentioned in Chapter 1, aggregate data structures are used extensively in system level C code. Here, we use the earlier example of the protocol stack to illustrate how we can find the relationship between two disjoint threads of execution. Consider the code fragment in Figure 2.3. This function is usually called from a network interrupt handler in the Linux kernel when a packet is received by a network interface card. It adds a reference to the net_device structure to the per cpu softnet_data poll list and schedules the execution of a softirq, which usually marks the end of the interrupt handler for a device. The rest of the packet processing happens in the context of a softirq. A kernel developer trying to understand network packet reception in the Linux code would be interested in knowing where exactly this happens. A good strategy would be to look for places where the net_device structure is being taken out
of the softnet.data poll list. CVision allows the user to specify exactly this event as a query, thus taking the user precisely to the point where the further processing of the packet begins.

```c
static inline void _netif_rx_schedule(
    struct net_device *dev)
{
    unsigned long flags;
    local_irq_save(flags);
    dev_hold(dev);
    list_add_tail(&dev->poll_list,
        &__get_cpu_var(softnet_data).poll_list);
    if (dev->quota < 0)
        dev->quota += dev->weight;
    else
        dev->quota = dev->weight;
    _raise_softirq_irqoff(NET_RX_SOFTIRQ);
    local_irq_restore(flags);
}
```

Figure 2.3 Function _netif_rx_schedule()

For large software, visualizations such as call graphs can be very complex with hundreds of nodes and thousands of edges. This complexity often defeats the purpose of visualization [8]. We propose graph reductions as way to manage such complexity and facilitate program comprehension. The tracking of events can be combined with graphical representations of relationships between program elements to produce highly useful event views. For example, instead of generic call graphs, we can produce event views that focus on flow relationships between events such as allocation and deallocation of memory, wait and signal on semaphores, locking and unlocking of locks etc. This is discussed in the next section.

### 2.2 Graph Reductions

When using the top-down strategy, a developer first constructs macro-level views of the code that capture cross-cutting relationships among functions and global data structures. For example, a call graph provides a pictorial view of function calls starting with a given function as the root. In this chapter we discuss graphical representations and their reductions as supported by CVision. CVision allows the user to perform these reductions interactively. A code comprehension strategy may not follow a preset trajectory. It may very well be a process...
of iterative refinement where the refinement depends on the intermediate results from applying the tool. CVision allows the user to incrementally refine the graphical representations. This lets the user develop multi-step strategies to arrive at the points of interest.

CVision supports three different types of graphical representations: call graph (CG), reverse call graphs (RCG), and control flow graph (CFG). These are well known graphical representations of a program. The call graphs and the reverse call graphs provide a macro-level view of the program flow that focuses on cross-cutting relationships among functions. The control flow graph focuses on micro-level view of the program flow within individual functions. We will discuss how the macro and micro level views and event views can be integrated through graph reduction techniques to produce a highly effective program comprehension tool.

With large programs generic macro-level views can easily get too large to be useful. For example, the call graph of `ip.rcv()`, which is the main IP receive function in the Linux kernel, has 1068 nodes and 2663 edges. The reverse call graphs are useful to understand interaction between multiple threads but they tend to be even larger. A common technique used by existing tools [20, 16] to manage this complexity is to incrementally unfold a graph upon user request. Only a few levels in a graph are shown to the user at a time. However, the number of levels can easily run into hundreds making incremental unfolding very tedious and not very useful. It does not allow the user to look deeper into the graph without going through a series of interactions. Additionally, incremental unfolding does not work well if the same function is at different levels on different call chains. Next, we discuss various types of graph reductions as a way to make graph representations useful for large code.

2.2.1 Control Flow Based Reduction

The objective of this technique is to reduce graphical representations of code based on control paths of interest. CVision allows the user to interactively exclude control paths from future analysis. Such a refinement eliminates parts of the call graph associated with the excluded control paths. The next example illustrates this reduction.

Consider the code in Figure 2.5. It is for the network input daemon in the Xinu operating
system. The body of the function is a forever loop that reads incoming packets and processes them. Based on the packet type, the switch statement transfers control to the appropriate protocol handler. When tracking incoming IP packets, a developer going through this piece of code would like to ignore all but one case of the switch statement. In this scenario, the user can apply the control flow based graph reduction to construct a customized call graph of netin(). After the reduction, arp_in() and rarp_in() do not show up in the customized call graph of netin().

2.2.2 Function Based Reduction

Consider the schedule() function in the Linux kernel. Linux developers know that a call to schedule() preempts the current thread of execution and reassigns the CPU to a different task in the system. A developer who is trying to analyze a piece of code for memory leaks, for example, would not be interested in the schedule(). CVision provides an interface
Figure 2.5  Code for function netin() - Network input daemon in Xinu

through which the user can specify functions to exclude from subsequent analysis. Macro-level views can be greatly simplified by excluding analysis of functions not relevant to the given concern. For example, excluding just the two functions schedule() and panic(), the call graph for tcp_v4_init() function in Linux goes down from 846 nodes to 653 nodes. Typically, a significantly large number of functions in a call graph are irrelevant given a specific concern and thus a function based reduction can be quite useful.

When the call graph contains more than a threshold number of functions, CVision brings up a list of functions in the call graph. Based on domain knowledge and understanding of the nature of the problem at hand, the user can select functions to be excluded from the call graph. Function based reduction is an iterative process that depends on the domain knowledge of the user. Users with good understanding of the domain are likely to get better results quickly compared to novice users who are completely new to the domain. To help novice users, it is
useful to have a list of functions (e.g. `printk()` in Linux) that can be safely omitted for many concerns.

Next, we describe two other types of reductions which provide the user powerful ways of specifying reductions without getting burdened with the task of coming up with an explicit list of functions to be omitted from subsequent analysis.

### 2.2.3 Code Layout Based Reduction

The layout of the source code is a good indicator of modularity in the code. For example, in Linux, all the networking code can be found in the `net/` directory. Further, the data types and data structures used by functions can be used as a good way to characterize the subset of functions that need to be included (or excluded) for an event analysis. For example, in Linux, functions that use `struct sk_buff` belong to the networking subsystem. In well written C code data types and data structures are usually declared in header files. Thus, inclusion (using `#include`) of certain header files can be a good criterion to perform graph reductions.

Consider the reverse call graph of `kmalloc()`, a widely used function in the Linux kernel. The complete reverse call graph of `kmalloc()` is very large. Typically, a user would be interested in a subgraph that corresponds to a particular subsystem. CVision lets the user reduce the reverse call graph based on code layout. For example, the user can create subgraphs based on the code layout information such header files or directories used to organize the source code files.

### 2.2.4 Event Based Reduction

This is the most flexible and powerful technique for graph reductions. Resource utilization (e.g. memory usage) or the use of a global data structure are examples of cross-cutting events that are entangled with each other in a complex body of code. It is very valuable if a tool can help the user in constructing macro-level views that isolate and emphasize a given event. We propose the event based reduction as a mechanism to create such focused macro-level views. In this reduction, a macro-level view is refined iteratively to focus on a set of events important
for the given concern. The next example provides an illustration.

Call graphs can be cluttered with functions that are irrelevant to the specific behavior of the program that the developer is interested in. For example, consider the code in Figure 2.6. The function `dswrite()` allocates memory for `drptr` which is a pointer of type `struct dreq` but does not free it. This implies a potential memory leak. Let us suppose the concern is to verify if there is a memory leak in this code. The call graph of `dswrite()` shown in Figure 2.4(a) includes all the function calls many of which are not relevant for analyzing the possibility of a memory leak. We can use an event based graph reduction to eliminate the unnecessary part of the call graph. The details are discussed later as a case study. The result of event based graph reduction is shown in Figure 2.4(b). By applying just one event based reduction, the original graph with 29 nodes is reduced to the event-specific call graph shown in Figure 2.4(b) with only 4 nodes. CVision lets the user specify “call to `freebuf()`” as the event of interest. Internally, CVision uses a graph reachability algorithm to compute the subgraph that retains only those nodes and edges from the original call graph that reach the `freebuf()` node.

```c

dwrite(devptr, buff, block)
struct devsw *devptr;
char *buff;
DBADDR block;
{
 struct dreq *drptr;
 char ps;
 disable(ps);
 drptr = (struct dreq *) getbuf(dskrbp);
 drptr->drbuf = buff;
 drptr->drdca = block;
 drptr->drpid = currpid;
 drptr->drop = DWRITE;
 dskenq(drptr, devptr->dvioblk);
 restore(ps);
 return(OK);
}
```

Figure 2.6 Code for function `dwrite()` - Disk write function in Xinu

CVision allows the user to refine the macro-level views based on the micro-level events listed in Table 2.1. Chapter 4 gives comprehensive examples of all the reduction techniques discussed so far.
CHAPTER 3. CVision Architecture

Figure 3.1 shows the architecture of CVision. It is a client-server architecture. The *backend* extracts program information on events, control flow, source correspondence etc. This information is stored in a relational database. The client does query processing and visualization. It uses the database to provide the user an interactive capability to produce call graphs, reverse call graphs, control flow graphs and customized event views. It also allows the user to see the source code in conjunction with visual representations.

The layered architecture of CVision is designed to provide efficiency, extensibility and flexibility. It permits independent modifications of the backend and the client as long as they adhere to the structure of the information database. Moreover, the database schema is independent of the programming language being analyzed and can be used for any structured programming language. Thus, it is possible to build a tool like CVision for other structured programming languages by just changing the backend.

![Figure 3.1 Architecture of CVision](image-url)

Figure 3.1 Architecture of CVision
3.1 The Backend

The backend is a language-specific semantic event analyzer. We have currently developed a backend for the C programming language. The C backend is an extension of the EDG compiler front-end. We process the parse tree generated by EDG and extract the following type of program information.

- **Program artifacts information.** We extract information about program artifacts like functions, statements, variables, types etc. The relationships between these program artifacts is also extracted, for example, which function a given statement belongs to, what is the type of a given variable etc. The goal is to pre-compute and store a rich set of semantic information. This information is used for characterizing events. Because of the pre-computation, the client part can be lightweight and fast.

- **Control flow information.** Each basic block is assigned a unique identifier. The control flow links to immediate predecessors and successors are stored in the database. Based on these links, the client can easily generate the control flow graph of a function.

- **Event information.** We extract information about the events as described in Table 2.1. The information is used for creating event views.

- **Source Correspondence.** For every event and artifact, the database stores the corresponding source location information. This is used for navigating through the code in conjunction with the graphical representations.

- **Auxiliary Information.** Since CVision is an interactive tool, fast response time is critical. We store auxiliary information to make subsequent analysis fast. For example, it is time consuming to determine all the call graphs to which a function belongs. This information is pre-computed and stored in the database. Subsequently, the information is used by the client for a fast computation and reduction of reverse call graphs.
3.2 The Information Database

We use MySQL database to store the information extracted by the backend. The schema of the database is independent of the programming language. In this section we describe the various tables of the database.

3.2.1 File Table

Source correspondence information associated with all the artifacts and events is stored in the database. To avoid repeating the file name in multiple tables of the database, we store the fully qualified file name of the source files in the file table. When referring to a file, all the other tables use the file identifier instead of the file name. Table 3.1 shows the file table.

<table>
<thead>
<tr>
<th>File Id</th>
<th>File Name</th>
</tr>
</thead>
</table>

3.2.2 Function Table

The function table stores the information about every function in the source project. Table 3.2 shows the structure of the function table in the database. Apart from storing source correspondence and semantic information about the function, we store some additional information about the function which helps the client in providing real-time interactive experience to the user. Based on the number of times the function is called, the number of distinct call graphs the function belongs to and the number of functions in its call graph, the client caches information regarding the most widely used functions to speed up the analysis.

3.2.3 Block Table

The block table stores information about the basic blocks in the source code. Along with the source correspondence information the table stores the identifier of the function the block
belongs to, the number of statements in the block, the type of the block (for e.g. if-block, while-block etc) and the block label if any. Table 3.3 shows the block table.

### 3.2.4 Control Flow Table

The edges between basic blocks in the code are stored in the control flow table shown in Table 3.4. We store the from block and the to block the edge connects. Additionally, we store the type of the edge in the table. The edge type could be a then-edge, else-edge, loop-edge etc.
3.2.5 Statement Table

The statement table stores information regarding the statements in the code. Apart from the usual source correspondence, type etc we store the complexity of the statement in the table. This helps in correlation of conditions. Two conditions connected by control flow are correlated if they are simple and use the same variables in the test condition. We define two metrics for complexity: arithmetic complexity and logical complexity. The logical complexity measures the number of logical operations like logical-AND (&&) used in the statement. The arithmetic complexity measures the number of arithmetic operations in a statement. Table 3.5 shows the format of the statement table.

Table 3.5 Statement table

<table>
<thead>
<tr>
<th>Unique Function Block Id</th>
<th>Statement</th>
<th>Statement Start Line</th>
<th>Statement Start Column</th>
<th>End Line</th>
<th>End Column</th>
<th>Arithmetic Complexity</th>
<th>Logical Complexity</th>
</tr>
</thead>
</table>

3.2.6 Function Call Table

The function call table stores the function call information in the source code. This is helpful in building various graphical representations of the code discussed in Section 2.2. Since the number of parameters passed in a function call varies at every call site for functions that take variable parameters, we have a column for the number of parameters passed at a particular call site. The function call table is shown in Table 3.6.

3.2.7 Variable Table

All information associated with a variable is stored in the variable table shown in Table 3.7. We store the declaration and definition information of a variable. We discuss the use of this information in Section 3.3.2. The Init Kind column contains the initialization information
for the variable. If a variable is being initialized, the place of definition of the variable is also a place of modification.

3.2.8 Type Table

Table 3.8 shows the type table. It holds information about the types in the source code. The type kind column tells us whether the type is built-in or user-defined.

3.2.9 Type Association Table

The type association table associates a particular variable to a type. The reason we define a separate table to store this information is to allow associating multiple types to a variable.
For example, an integer variable $x$ which is a member of struct $s1$ can be viewed as being of type $\text{int}$ as well as of type $s1.x$. This table is currently unused.

<table>
<thead>
<tr>
<th>Table 3.9 Type association table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique Association Id</td>
</tr>
</tbody>
</table>

### 3.2.10 Read-Write Table

The read-write table shown in Table 3.10 stores the read/write information associated with a variable. It connects the read-write operation with a variable, a statement and a function in the source code.

<table>
<thead>
<tr>
<th>Table 3.10 Read-Write table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique Read-Write Id</td>
</tr>
</tbody>
</table>

### 3.2.11 Parameter Table

The parameter-passing information is stored in a parameter table. The structure of the parameter table is shown in Table 3.11. A row in this table tell us which variable is being passed as a parameter to a function at a particular call site.

### 3.2.12 Function Summary Table

The backend calculates statistical information about each function and stores it in the function summary table. The structure of the table is shown in Table 3.12.
### Table 3.11 Parameter table

<table>
<thead>
<tr>
<th>Unique Parameter Id</th>
<th>Calling Function Id</th>
<th>Function Call Id</th>
<th>Variable Id</th>
<th>Parameter Number</th>
</tr>
</thead>
</table>

### Table 3.12 Function summary table

<table>
<thead>
<tr>
<th>Function Id</th>
<th>Number of Conditions Using Only Local Variables</th>
<th>Number of Conditions Using Only Global Variables</th>
<th>Number of Conditions Using Only Parameters</th>
<th>Number of Conditions Using Only Internal Types</th>
<th>Number of Conditions Using Only User Defined Types</th>
<th>Number of Conditions Using Local Variables</th>
<th>Number of Conditions Using Global Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
<td>-------------------------------------------------</td>
<td>--------------------------------------------</td>
<td>-----------------------------------------------</td>
<td>---------------------------------------------</td>
<td>-------------------------------------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>Number of For Loops</td>
<td>Number of For Loops</td>
<td>Number of For Loops</td>
<td>Number of For Loops</td>
<td>Number of For Loops</td>
<td>Number of For Loops</td>
<td>Number of For Loops</td>
<td>Number of For Loops</td>
</tr>
<tr>
<td>Number of Do Loops</td>
<td>Number of Do Loops</td>
<td>Number of Do Loops</td>
<td>Number of Do Loops</td>
<td>Number of Do Loops</td>
<td>Number of Do Loops</td>
<td>Number of Do Loops</td>
<td>Number of Do Loops</td>
</tr>
<tr>
<td>Number of While Loops</td>
<td>Number of While Loops</td>
<td>Number of While Loops</td>
<td>Number of While Loops</td>
<td>Number of While Loops</td>
<td>Number of While Loops</td>
<td>Number of While Loops</td>
<td>Number of While Loops</td>
</tr>
<tr>
<td>Number of If Statements</td>
<td>Number of If Statements</td>
<td>Number of If Statements</td>
<td>Number of If Statements</td>
<td>Number of If Statements</td>
<td>Number of If Statements</td>
<td>Number of If Statements</td>
<td>Number of If Statements</td>
</tr>
<tr>
<td>Number of Switch Statements</td>
<td>Number of Switch Statements</td>
<td>Number of Switch Statements</td>
<td>Number of Switch Statements</td>
<td>Number of Switch Statements</td>
<td>Number of Switch Statements</td>
<td>Number of Switch Statements</td>
<td>Number of Switch Statements</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Conditions Using Parameters</th>
<th>Number of Conditions Using Internal Types</th>
<th>Number of Conditions Using User Defined Types</th>
<th>Number of Simple Conditions</th>
<th>Number of Logically Simple Conditions</th>
<th>Number of Correlatable Conditions</th>
<th>Number of Correlation Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Blocks</td>
<td>Number of Stmnts</td>
<td>Number of Blocks In COG</td>
<td>Number of Stmnts In COG</td>
<td>Number of Functions In COG</td>
<td>Preprocessed</td>
<td></td>
</tr>
</tbody>
</table>
3.2.13 Correlation Table

Correlatable conditions are stored in the correlation table shown in Table 3.13. The block numbers of the correlatable test condition are stored in the table. Correlation information is intra-procedural. The conditions belonging to the same correlation group are all mutually correlatable.

Table 3.13 Correlation table

<table>
<thead>
<tr>
<th>Unique Correlation Id</th>
<th>Function Id</th>
<th>Correlation Group</th>
<th>First Block Number</th>
<th>Second Block Number</th>
</tr>
</thead>
</table>

3.2.14 Function Root Table

The function root table shown in Table 3.14 stores information which is used in preprocessing, caching and code layout based graph reductions described in Section 2.2.3. Each entry in this table gives us the identifier of the root function of a call graph a particular function belongs to. For example, if function f2 is in the call graph of function f1 and f1 is a root function, f2, f1 is stored in this table.

Table 3.14 Function root table

<table>
<thead>
<tr>
<th>Unique Id</th>
<th>Function Id</th>
<th>Root Function Id</th>
</tr>
</thead>
</table>

3.2.15 Include Hierarchy Table

Include hierarchy table contains the #include hierarchy in the source code. This helps in code layout based reductions discussed in Section 2.2.3.


3.3 The Client

The client interacts with the database server and the source code to implement graph reduction techniques and construct event views.

The user interface for CVision has been designed to provide an integrated environment for code navigation and comprehension. It supports textual and graphical views of the code. It provides user interfaces for macro and micro level views. CVision integrates these different features so that the user can seamlessly switch between views and level of abstractions.

CVision provides textual views of the code along with the graphical views described in Section 2.2. The user can click on any node of a graphical representation and view the source code associated with it.

CVision integrates the control flow graphs with call graphs and reverse call graphs. For example, the user can click on a node in the call graph and view the control flow graph of the corresponding function. When a call graph is reduced based on events, the control flow graph nodes of all the functions in the call graph are colored to show locations of the event. The use of such coloring is brought out in the case study in Section 4.1. Additionally, when the user decides to omit a particular path in the control flow graph of a function from future analysis, the call graph for the function reflects this choice by eliminating all the nodes that are reachable only through omitted paths.

The client provides views based on functions, variables and types in a program. The rest of this section gives the implementation details of these features.
3.3.1 Function Views

The function view provides information about a particular function. The interactive user interface lets the user query for specific events in the function. The user can get information regarding the variables and types used in the function. The user can generate call graph, reverse call graph and control flow graph of the function and perform graph reductions described in Section 2.2 on these graphs.

Figure 3.2 shows the user interface of the function view for the dskenq() function. It shows the list of functions that are called by dskenq(). The user can choose one of these functions and view its call site or bring up its function view. Similarly it show a list of functions that call dskenq(). This helps the user to move up or down the call graph one level at a time.

Figure 3.2 User interface for function views
The function view allows the user to interactively query for specific use of different kinds of variables and types. The user can switch to a variable or type view by clicking on a variable or type from the result list.

Figure 3.3 shows the interface for the graphical representations of the function. The interface allows the user to incrementally reduce the graph and to move between successive reductions. Figure 3.4 shows the user interface for applying reductions on the graphs. CVision provides the same interface for graph reductions on all graphical representations of the code. This interface provides the user a single window to apply all the reductions discussed in Section 2.2. Alternatively, in a call graph, the user can click on a particular node in the graph and apply function based reduction. For control flow graphs we allow the user to choose an edge in the graph and apply flow based reduction on the graph.

3.3.2 Variable and Type Views

The variable view provides information about a particular variable. Using this view the user can query for specific events involving the variables. For example, the user can ask for all the places where a particular variable is being passed as a parameter to a specific function. Variable views are particularly helpful in tracking the use of global variables and aggregate data structures.

Figure 3.5 shows the user interface of the variable view. The user can query for specific events based on definition or declaration. For example, consider the code in Figure 3.6. Here the variable \( i \) is declared as a member of the structure \( s \), but is being defined in \( foo() \) and \( bar() \). If the user makes a query for all the places where \( i \) is being modified based on its declaration, the relevant statements in \( foo() \) and \( bar() \) will be shown. If the user makes the same query based on the definition, only one of the two statements will be shown based on which particular instance of \( i \) the user is interested in, \( a.i \) or \( b.i \).

The type view provides a similar interface for built-in and user-defined types. Figure 3.7 shows the user interface for the type view.
Figure 3.3 User interface for graphical representations of the code

3.4 Evolution and Performance

CVision currently has more than 10000 lines of code which was developed over a period of 12 months. The design and feature set of CVision has continuously evolved during this period. The motivation for this continuous enhancement is two-fold: effectiveness and performance. In this section, we briefly discuss the evolution process and the motivations.

3.4.1 Effectiveness

A considerable amount of time was spent in identifying the modus operandi of a developer trying to understand large and complex code. We then sought to automate this process by
Figure 3.4 User interface for graph reductions

incorporating various features in CVision. For example, when looking at a complex function, we found that the developer mentally ignores certain execution paths. To map this strategy to a feature in CVision, we incorporated control flow based graph reduction.

Additionally, using our experience in defect analysis and code comprehension we identified the events of interest for various code comprehension activities. Based on these events, we incorporated event views in CVision.

Many features were incorporated late in the development cycle. For example the code layout based reduction was developed after we found that other graph reduction techniques are not effective with reverse call graphs.

The features provided by CVision provide building blocks using which the developer can
compose and execute a code comprehension plan.

3.4.2 Performance

Performance was a significant concern during the design and development of CVision because of two reasons. Firstly, since CVision is an interactive tool, real-time response to user requests is crucial. Secondly, scalability is essential in order to handle large code. As mentioned in Chapter 1, system level C code usually runs into millions of lines. In this section we discuss the various strategies we use to improve the performance of CVision.
3.4.2.1 Preprocessing

We preprocess the information extracted from the source code before storing it into the database. Consider the following example. CVision uses the control flow information of a function for various purposes. Further, the control flow information does not change unless the source code for the function changes. We pre-calculate the control flow information of each function based on the statements in the function and store it into the database. Therefore, the client does not have to recalculate this information every time it is needed. This reduces the processing done by the client and increases the responsiveness of CVision.

Additionally, we pre-calculate and store some statistical data about the code. This helps in caching (discussed in Section 3.4.2.2) and graph reductions. The statistical data is stored in the Function Table and the Function Summary Table described in Section 3.2.2 and Section 3.2.12 respectively.

3.4.2.2 Caching

The client processes a large amount of information spread across the source files. Querying the database everytime some information is required adversely affects the response time of the client. For example, when performing graph reductions, the client needs event information

```c
struct s {
    int i;
};

int foo()
{
    struct s a;
    a.i = 100;
}

int bar()
{
    struct s b;
    b.i = 100;
}
```

Figure 3.6 Sample code
from all the nodes in the graph. As explained in Section 2.2, for a large code base the number of nodes in the graph can easily run into thousands. Fetching this information from the database on demand can take several seconds. To overcome this problem, the client maintains a LRU cache which contains the information retrieved from the database. This enables the client to reuse the results of previous queries. Each cache entry stores all the information associated with a particular function. When information about a given function is needed, the client executes a database query only if the information is not already available in the cache.

The cache is populated in the following two ways.

1. Prefetching
Each time a new project is opened, the client prefetches information associated with functions that meet a prefetch criterion. If a function meets the prefetch criterion, then all the functions in its call graph are prefetched too. The prefetch criterion is as follows:

*We calculate the average number of functions in the call graphs of the project. If the number of functions in the call graph of a function exceeds this average, it qualifies for prefetching. We then sort the qualifying functions in the decreasing order of the number of distinct call graphs each function belongs to. The first 2000 functions in this sorted list are prefetched along with all the functions in their call graph.*

The prefetching is done via background threads so that the client can continue to process other user requests.

2. *On Demand*

Each time information associated with a particular function is needed, the information is fetched from the database and the cache is populated with this information. Additionally, we prefetch information associated with all the functions in the call graph of the function.

We have observed a significant performance improvement due to the cache. Operations that used to take 30-60 seconds without the cache can be performed in less than 10 seconds with the cache.

3.4.2.3 Query Optimization

The information database is indexed to speed up queries. The columns used in the indexing are based on the most frequent queries executed by the client. Table 3.16 shows the primary keys and columns used for indexing.
Table 3.16  Primary keys and indices

<table>
<thead>
<tr>
<th>Database Table</th>
<th>Primary Key</th>
<th>Indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Table</td>
<td>File Id</td>
<td>(File Name)</td>
</tr>
<tr>
<td>Function Table</td>
<td>Function Id</td>
<td>(Function Name, Declared File Id, Declared Line Number), (Number of times called), (Declared File Id)</td>
</tr>
<tr>
<td>Block Table</td>
<td>Block Id</td>
<td>(Function Id, Block Number)</td>
</tr>
<tr>
<td>Control Flow Table</td>
<td>Flow Id</td>
<td>(Function Id, From Block Id)</td>
</tr>
<tr>
<td>Statement Table</td>
<td>Statement Id</td>
<td>(Function Id, Block Id, Statement Number)</td>
</tr>
<tr>
<td>Function Call Table</td>
<td>Call Id</td>
<td>(Calling Function Id), (Called Function Id), (Calling Function Id)</td>
</tr>
<tr>
<td>Variable Table</td>
<td>Variable Id</td>
<td>(Variable Name, Defined File Id, Declared File Id, Defined Line Number, Declared Line Number)</td>
</tr>
<tr>
<td>Type Table</td>
<td>Type Id</td>
<td>(Type Name, Declared File Id, Declared Line Number, Declared Column Number)</td>
</tr>
<tr>
<td>Type Association Table</td>
<td>Association Id</td>
<td>(Variable Id), (Type Id)</td>
</tr>
<tr>
<td>Read Write Table</td>
<td>Read Write Id</td>
<td>(Function Id, Variable Id, Read Write), (Statement Id), (Variable Id, Read Write)</td>
</tr>
<tr>
<td>Parameter Table</td>
<td>Parameter Id</td>
<td>(Function Call Id, Parameter Number), (Variable Id), (Calling Function Id)</td>
</tr>
<tr>
<td>Function Summary Table</td>
<td>Function Id</td>
<td>None</td>
</tr>
<tr>
<td>Correlation Summary Table</td>
<td>Correlation Id</td>
<td>(Function Id, Correlation Group)</td>
</tr>
<tr>
<td>Function Root Table</td>
<td>Function Root Id</td>
<td>(Function Id), (Root Function Id)</td>
</tr>
<tr>
<td>Include Hierarchy Table</td>
<td>Include Id</td>
<td>(File Id, Included File Id), (Included File Id)</td>
</tr>
</tbody>
</table>
CHAPTER 4. Case Studies

The objective of the case studies is to illustrate how CVision helps the user in overcoming following challenges for program comprehension:

- **Large code base.** We show how CVision helps users apply his domain knowledge to distill large code by filtering irrelevant information and focusing on parts important for their concerns.

- **Concurrent processing.** We show how CVision helps users connect two disjoint threads of execution and understand their inter-relationships.

- **Runtime Bindings.** We show how CVision helps users decipher runtime bindings information.

- **Aggregate data structures.** We show how CVision helps users track the cross-cutting use of aggregate data structures.

The first case study is about a defect analysis problem encountered in the Xinu operating system. The second case study is about understanding the TCP/IP code in Linux.

The purpose of case studies is to show applicability of CVision in solving non-trivial program comprehension problems encountered in real-world maintenance applications. The case studies focus on main points and purposely omit some details that can be distracting. Problems similar to the case studies were given as maintenance projects in two software engineering courses taught in the Electrical and Computer Engineering Department at Iowa State University [7, 6]. The case studies as presented here are similar to important points covered in lectures. The objective is to cover important aspects of how to apply the event views and graph reductions
in solving a complex problem. In our courses, students learn the details and come up with a complete solution to the case study problems through an experimental study using CVision.

### 4.1 A Xinu Case Study

In this case study we take a closer look at the possibility of memory leak in the \texttt{dswrite()} function of Xinu operating system. Figure 2.6 shows the code for \texttt{dswrite()}. As explained earlier in Section 2.2, memory is allocated for \texttt{drptr}, which is a pointer to the type \texttt{struct dreq}, but never freed inside \texttt{dswrite()}. \texttt{drptr} escapes the function through a call to the function \texttt{dskenq()}. Thus, it is possible that The memory allocated for \texttt{drptr} gets freed elsewhere in the system. If not, we have a memory leak. Let us investigate this problem further.

1. **We reduce the call graph of \texttt{dswrite()} from 29 functions to 4 functions using an event based graph reduction.**

   Figure 2.4(a) shows the full call graph of \texttt{dswrite()}. It has 29 functions many of them are irrelevant to the analysis of the memory leak problem. We apply an event based reduction on the call graph with “call to \texttt{freebuf()}” as the event of interest. This reduction retains only that part of the call graph where \texttt{freebuf()} is reachable. Figure 2.4(b) shows the reduced call graph. Note that this is the part of the call graph that needs to be analyzed for solving the memory leak problem.

2. **We observe the colored control flow graphs to locate the \texttt{freebuf()} events.**

   The reduced call graph is shown in Figure 2.4(b). We follow the call chain and track the pointer (This can be done by specifying another event involving parameter passing; the detail is not covered here). Looking at the call graph we know that a call to \texttt{dskenq()} eventually leads to a call to \texttt{freebuf()}. This is not enough to rule out the possibility of memory leak because there can still be a control flow path which does not have a call to \texttt{freebuf()}. This implies we have to look inside functions and observe where the events occur on the control flow paths. Earlier we have mentioned the coloring feature of CVision; it becomes useful here. Whenever an event based reduction is applied
to a call graph, CVision colors those code blocks (shown as nodes in the control flow graphs) of all the functions in the reduced call graph. The colored blocks are exactly the locations pertaining to the given event. In this particular example, the code blocks in `dskenq()` that call `dskqopt()` appear colored in the control flow graph of `dskenq()` because `dskqopt()` calls `freebuf()`. Thus, by looking at the colored control flow graph we can quickly isolate the important code blocks. Figure 4.1 shows the code for the colored block which in this case is the one that calls `dskqopt()` (which subsequently calls `freebuf()`). The colored control flow graph of `dskqopt()` shows multiple places where `freebuf` is called. These code for these blocks are shown in Figure 4.2.

3. We observe a path in the colored control flow graph `dskqopt()` that does not include a call to `freebuf()`.

A quick pass through the control flow graph of `dskqopt()` reveals that there is a path that does not have a call to `freebuf()`. Now, we take a close look at the code on that particular path. The code is shown as the last piece of code in Figure 4.2. We see that `drptr` is being added to a linked list without being freed. (At this point, we can also track the condition governing specific path and observe that it correlates with the condition in `dswrite()`; again these details are not covered here). Since `dskqopt()` is the only function in the reduced call tree that calls `freebuf()`, we know for sure that the memory allocated for `drptr` in `dswrite()` is not being freed along the path we have discovered. However, the analysis is not complete yet. Since `drptr` is being added to a linked list, it is possible that it is being freed by another disjoint thread of execution. Assuming such a thread exists, the next step is to discover it.

4. We unearth the relation between two disjoint threads of execution in Xinu using a type based event view.

Note that we have reached a dead end if were to use only call graphs for solving the memory leak problem. A call graph is based on control flow and two disjoint threads of execution are not connected by control flow. We now apply a different strategy.
Two disjoint threads of execution communicate through a shared data structure. The linked list to which \texttt{drptr} is being added is such a shared data structure. Based on this knowledge, we execute an event based query on the entire source code. The event we look for this time is "\textit{passing of the type struct \texttt{dreg} as a parameter to \texttt{freebuf()}". CVision comes up with 8 locations for this event in the entire code for Xinu. Four of these are in \texttt{dskgopt()} and we have already looked at these. Two others are in \texttt{dsread()} and \texttt{dsksync()}. The one more place that CVision comes up is in \texttt{dsint()}.

Figure 4.3 shows the code for \texttt{dsint()}.

We see that there is a call to \texttt{freebuf()}. Note that we have additional evidence such as \texttt{dsint()} is a root function (not called by any other function) and it calls \texttt{freebuf()} but not \texttt{getbuf()} indicating the matching \texttt{getbuf()} is on another thread of execution. We can also correlate the conditions under which \texttt{getbuf()} is called in \texttt{dswrite()} and conditions under which \texttt{freebuf()} is called in \texttt{dsint()}.

5. Using domain knowledge, we consolidate and summarize our understanding.

Applying domain knowledge and the key details of the code discovered above, a system developer could conclude: \texttt{dswrite()} allocates memory for \texttt{drptr}. Since this is a write operation to the disk, \texttt{dswrite()} adds \texttt{drptr} to a linked list and schedules a write to the disk. When the disk controller interrupts the CPU, marking the successful end of write operation, the interrupt handler \texttt{dsint()} frees the memory allocated for \texttt{drptr}.

Without a tool such as CVision, comprehending the above memory leak problem is hard.
Figure 4.2 Code for function dskqopt()

It involves going through a large body of code and tracking complex interactions that happen not only across multiple functions but across disjoint threads of executions.

4.2 A Linux Case Study

The comprehension problem for this case study is as follows:

Preamble: As explained in the introduction, when a network packet arrives at an interface, the network interface card transfers the packet from the device memory to the kernel memory and interrupts the CPU. The interrupt handler does the preliminary processing of the interrupt and leaves the rest of the processing for kernel threads.

Problem: Given the code for the Intel E1000 ethernet device driver, the problem is to discover
the exact details of packet reception including the execution threads, functions, data structures, and their relationships as incorporated by the Linux kernel.

1. We apply graph reductions to reduce the call graph of e1000_intr() from 824 functions to 7 functions.

We start off with the call graph of e1000_intr(). The call graph is not shown because it is too complex; it has 824 nodes and 1964 edges in it. We apply function based graph reductions described in Section 2.2 to reduce the size of the graph. Functions like schedule(), panic() and printk() are clearly not of interest for solving the given problem. Similarly there are many other functions that are irrelevant to our problem and can be omitted from further analysis. The reduced call graph is shown in Figure 4.4; it has only 7 nodes and 6 edges.

2. We find a data structure that can be used to discover the relevant kernel thread.

We view the code for _netif_rx_schedule() shown earlier in Figure 2.3. The code reveals that this function puts the net_device structure on a per cpu poll list in the softnet_data data structure and schedules the execution of the NET_RX_SOFTIRQ softirq. As mentioned earlier, this marks the end of the E1000 interrupt handler. Rest of the processing of the packet must happen in another thread of execution. We need to discover
that other thread of execution.

The Figure 4.5 shows an high-level view of the type of connection between an interrupt handler and a kernel thread that we are trying to discover. We started from e1000.intr() and followed the control flow till the net_device structure is put on the poll list, which is an aggregate data structure. We know that some other thread of execution takes the net_device structure off the poll list and processes it, but we do not know the other thread of execution but we know the aggregate data structure it is expected to access.

3. **Using a variable based event view we connect two threads of execution.**

In step 2 we identified the data structure that binds the two threads of execution. At this point, we know that _netif_rx_schedule() puts net_device structure on to the poll_list. We now have to find out where it is being taken out of this list and processed. We specifically look for the code where poll_list member variable of the softnet.data structure is passed to list manipulation functions. Here, we are trying to use operations on aggregate data structures, in our case the linked list poll_list, to connect two disjoint threads of execution. Hence, we query for the places where the linked list poll_list is being passed as a parameter to a list manipulation function. Looking closely at the results of this query, we can see that the net_device structure is being taken out of the poll
list in the function `net_rx_action()`. Figure 4.6 shows the relevant parts of the code. By tracking the use of an aggregate data structure, we have discovered the interaction between two concurrent threads. We have tracked the processing of a network interrupt from the context of the interrupt handler to the context of the softirq.

4. We resolve a runtime binding.

We use variable based event views to find the functions that could be called using a particular function pointer in the `net_rx_action()` function. Figure 4.6 shows the source code for the function and Figure 4.7 shows the reduced call graph of the function. The node `poll` is colored to indicate that `net_rx_action()` makes an indirect call to a function using the function pointer `poll`. To locate the functions that could be called using the function pointer `poll`, we query for all the places in the code where the variable `poll` is written to. There are a few places in the Linux code where the `poll` member variable of the `struct net_device` is modified. Since we are currently concerned only with the E1000 ethernet device, we focus on the write event in the `e1000_probe()` function and see that `poll` is being set to the address of `e1000_clean()` function. We conclude that `e1000_clean()` is called using the function pointer `poll` in the function `net_rx_action()`.

---

Figure 4.5 Diagram depicting the use of aggregate data structures to link two disjoint threads of execution
static void net_rx_action(struct softirq_action *h) {
    ...
    while(!list_empty(&queue->poll_list)){
        struct net_device *dev;
        ...
        dev=list_entry(queue->poll_list.next, struct net_device, poll_list);
        ...
        if(dev->quota<=0||dev->poll(dev,&budget)){
            ...
        }
        ...
    }
}

Figure 4.6 Function net_rx_action()
CHAPTER 5. Related Work

Cscope [17] is a popular code navigation tool widely used in the industry. Cscope uses a symbol based cross-referencing to locate functions, function calls, macros, variables, and preprocessor symbols in the source code files. Cscope recognizes very few C language constructs. This limits its searching capability to only a few simple types of queries. In comparison, CVision has a significantly more powerful searching capability with a wide range of semantically rich queries. Unlike Cscope, it is aware of control flow and provides graphical views based on the control flow.

Understand for C++ [20] and Crystal [16] are semantic tools for code navigation and comprehension. They support various graphical representations of the code and use incremental unfolding to manage the size and complexity of these representations. As explained in Section 2.2, this method is not effective. The new techniques of event views and graph reductions that we introduce and incorporate in CVision are scalable and far more effective in solving complex program comprehension problems encountered with large system level software.

FEAT [14] introduces a notion of concerns for understanding large programs. Their notion of concerns focuses on structural relationships between program elements such “method x calls method y” or “class x defines method y”. In comparison, we provide a significantly broader notion of events. Apart from the foundational differences between the two approaches, CVision provides graphical representations and graph reduction techniques which are not in the FEAT tool. CVision is currently available for C whereas the FEAT tool is for Java.

The CodeSurfer [2] is another related tool. CodeSurfer provides graphical representations of control and data dependencies. CVision performs a comparatively lighter event analysis and supports users queries based on an information database created by the event analysis.
Further, CodeSurfer supports *Model Checking* which CVision does not. For handling complex program comprehension problems, a significant advantage of CVision is that it provides event views and graph reductions.

Other code comprehension tools like Imagix4D [19], ICICLE [15], ASSIST [12] etc and IDEs like Eclipse, Visual Studio etc provide limited support for queries. They do not have refined event definitions provided by CVision.

Tools like DMS [18] attempt automatic architecture recovery. CVision allows the user to derive such information interactively through various graphical representations of the code but stops short of automating this process. Our experience with program comprehension problems including the two case studies included in this thesis, suggests that the real world problems are so complex and variable that an interactive approach that takes advantage of human expertise is more likely to succeed in practice as opposed automated approaches with preset strategies.

Engler et al. [10, 9] perform a path-sensitive analysis to detect bugs in code. They do so by checking if all the paths of a program comply to programmer defined models and state machines. CVision does a path-sensitive analysis of the source code but the primary goal of CVision is program comprehension and not automated bug detection. The SLAM project at Microsoft [4] develops tools to check if a program obeys certain rules. SLAM uses static analysis to determine API usage violations in C programs. A theorem prover is used in SLAM to correlate conditions to reduce the number of false positives due to static analysis. In terms of defect analysis, CVision focuses on handling problems that involve complexities due to concurrent processing, use of aggregate data structures, and runtime bindings. Given the complexity of this type of defect analysis problems, it provides an interactive approach as opposed to static analysis tools that perform defect analysis in batch mode without human intervention.
CHAPTER 6. Conclusions

We have introduced two new techniques event views and graph reductions as way to cope with complex program comprehension problems involving large system level C programs. We discuss different types of events and various categories of graph reductions. We also discuss how event based program comprehension and graph reductions integrate with each other and together provide a powerful approach to the type of program comprehension that is necessary for maintaining large system level C programs. We describe CVision, a tool we have developed to demonstrate applicability of our approach. We present several short code examples derived from XINU and Linux operating systems to motivate various types of event views and graph reductions. Finally, we present two case studies based on complex program comprehension problems.

Our experiments with CVision have been very encouraging. The tool is found to be quite useful in working with Xinu and Linux operating systems. We believe that software engineers will find CVision like tools very useful. CVision has proven to be a very effective tool for educational purposes. It is being used to teach software maintenance in software engineering courses at Iowa State University [7, 6]. With help of the tool, students are able to do projects involving XINU and Linux codes, as opposed to toy problems with small code.

CVision has been developed in collaboration with EnSoft Corporation and efforts are under way to produce a commercial product.
BIBLIOGRAPHY


ACKNOWLEDGEMENTS

I would like to express my thanks to Dr. Suraj Kothari for his guidance, patience and support throughout this research and the writing of this thesis. His insight and clarity of thought were essential to the conception and implementation of CVision.

I would like to thank my committee members for their efforts and contributions to this work: Dr. Manimaran Govindarasu, Dr. Samik Basu and Dr. Rana Biswas.

I also thank Dr. Rana Biswas and the employees of IPRT Company Assistance for the support and patience during this work.

Finally, I would like to take this opportunity to thank Jason Stanek for his comments and suggestions. His help in improving the performance of the tool was invaluable.