AUTOMATED BUG LOCALIZATION IN EMBEDDED SOFTWARES

by

PARTHA PRATIM RAY

THESIS

Presented to the Department of Electronics and Communication Engineering

Haldia Institute of Technology

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF TECHNOLOGY

Department of Electronics and Communication Engineering

THE WEST BENGAL UNIVERSITY OF TECHNOLOGY

29^{th} April 2011

Acknowledgements

I would like to express my deep-felt gratitude to my adviser, Mr. Banibrata Bag, Assistant Professor of the Electronics and Communication Engineering department at Haldia Institute of Technology, for his advice, encouragement, enduring patience and constant support.

I thank Dr. Ansuman Banerjee, Assistant Professor of the Advanced Computing and Microelectronics Computing Unit of Indian Statistical Institute, Kolkata. He was extremely helpful in providing the keen guidance and expertise I needed in order to complete this work, especially with regard to the debugging aspect of embedded softwares. He was never ceasing in his belief in me, always providing clear explanations when I was lost, constantly driving me with energy when I was tired, and always, giving me his time, in spite of anything else that was going on.

I also wish to thank Professor Malay Kumar pandit and Associate Professor Mr. Asim Kumar Jana of the Electronics and Communication Engineering department, both at the Haldia Institute of Technology. Their suggestions and comments were invaluable to the completion of this work. Dr. Malay Kumar Pandit graciously volunteered to act as my thesis coordinator.

Additionally, I want to thank the faculties and staff of Electronics and Communication Engineering department for all their help to complete my degree and prepare for a career as a enthusiastic science-seeker. This includes (but certainly is not limited to) the following individuals:

Dr. Sunandan Bhunia

> He made it possible for me to have many wonderful experiences I enjoyed as a student, including the opportunity to get in touch with the external sources of knowledge base such as ISI, Kolkata. His believe and immense support on me and my work encouraged to move ahead in research work. I want to greet a very special thank to him.

Mr. Sourav Kumar Das

> His constant help and faith in Embedded Systems Laboratory during last few months provided me great support in completing my work.

And finally, I must thank my dear friends, father and mother, well wishers and obviously Poulami Majumder, for putting up with me during the development of this work with continuing, loving support and no complaint. I do not have the words to express all my feelings here, only that I love you, all!

Declaration

I hereby declare that this thesis entitled **"Automated Bug Localization In Embedded Softwares"** has not been accepted for any degree, diploma or course and is not currently submitted in candidature of any other degree, diploma or course.

Partha Pratim Ray

University Roll No. 09103069010

University Registration No. 091030410053

Date:

Certificate

This is to certify that the project entitled, **"Automated Bug Localization In Embedded Softwares"** submitted by Mr. Partha Pratim Ray in partial fulfillment of the requirements for the award of Master of Technology Degree in Electronics and Communication Engineering during session 2010-11 at the Haldia Institute of Technology is an authentic work carried out by him under our supervision and guidance. To the best of my knowledge, the matter embodied in the project has not been submitted to any other University or Institute for the award of any degree or diploma.

Dr.Ansuman Banerjee	Mr.Banibrata Bag
Assistant Professor	Assistant Professor
Indian Statistical Institute	Haldia Institute of Technology

Certificate of Approval

The foregoing thesis is hereby approved as a creditable study of an engineering subject carried out and presented in a manner satisfactory to warrants its acceptance as a pre-requisite for the degree for which it has been submitted. It is understood that by this approval the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein but approve the thesis only for the purpose for which it is submitted.

Committee for evaluation of the project:

Signature of Examiner	Signature of Examiner	Signature of Examiner
Date:	Date:	Date:

Abstract

Debugging denotes the process of detecting root causes of unexpected observable behaviors in programs, such as a program crash, an unexpected output value being produced or an assertion violation. Debugging of program errors is a difficult task and often takes a significant amount of time in the software development life cycle. In the context of embedded software, the probability of bugs is quite high. Due to requirements of low code size and less resource consumption, embedded softwares typically do away with a lot of sanity checks during development time. This leads to high chance of errors being uncovered in the production code at run time.

Debugging embedded softwares is a common problem. Several techniques and tools do exist for solving this problem. However, various kinds of bugs remain untouched regarding unauthorized and inefficient memory related issues and few bugs may also reside inside the states of the software code. No feasible algorithm or technique has ever been developed that will solve all the particular cases of this problem accurately.

It turns out, though, that the width of the debugging metrics are quite important in a large number of the embedded systems having various buggy issues. This becomes readily apparent when we learn that the bugs typically come from inaccurate debugging technique.

Any measurement of a buggy metric is limited by the precision and accuracy of the debugging technique. To be of practical use, the debugging process must be reasonably accurate. This implies that, for most of the parts, the actual values associated with measurements lie within relatively narrow interval parameters or metrics such as memory, invariant and object state. Indeed, manufacturers often guarantee the error of their softwares to be very small.

Thus, we desire to look only at narrow interval parameters when considering the development of the methodology for debugging embedded softwares. As no such effective techniques exist that can solve such software problems, developing such methodologies seems

indeed promising. Therefore, the goal of this thesis is to answer the following question:

> *Can any automated technique or methodology be developed for the general problem of debugging embedded softwares with narrow interval parameters?*

We show here that this problem of debugging embedded softwares with narrow interval coefficients, is quite possible; thus, we do consider it possible to develop a feasible technique that will solve all particular cases of this problem in automated way.

Table of Contents

List of Figures

Chapter 1

Introduction

Embedded softwares are the heart of embedded systems. The development of embedded softwares need to undergo the process of general software development cycle, resulting in code development and debugging as important activities. Embedded softwares typically do away with lot of sanity checks during development time. This leads to high chance of errors being uncovered in the production code at run time.

In this thesis we propose a methodology for debugging errors in embedded softwares. As a case study we have used BusyBox, de-facto standard for embedded Linux in embedded systems and a few pseudo codes resembling embedded BusyBox. Our first methodology works on top of Valgrind, a popular memory error detector. We second work deals with invariant analysis of some parts of BusyBox by Daikon, an invariant analyzer. As our final contribution we developed a technique which incorporates object state information into class dependence graph (cldg) to detect bugs in object oriented programs for embedded softwares.

Problem Statement : Different kinds of debugging techniques are available for debugging embedded softwares. Specific types of errors that result from memory mishandling, unauthorized memory access etc. are very much crucial to embedded softwares in respect of bug intrusion, that must be detected well at the time of development. Bugs can penetrate into softwares through other ways also. But the existing approaches to detect the above said bugs, are unable to execute its job satisfactorily. Hence the need for developing new debugging methodologies aroused.

Organization : This thesis is organized as follows. Chapter 2 presents literature review regarding various aspects of bug, debug, embedded systems and embedded software and their features. Chapter 3 presents the related work on this matter. Chapter 4 presents the memory error detection. Chapter 5 presents invariant analysis. Chapter 6 presents object state incorporated debugging technique. Chapter 7 presents future work and concluding remarks.

.

Chapter 2

Literature Review

2.1 Software Bug

A software bug [40] is an error, flaw, mistake, undocumented feature that prevents it from behaving as intended (e.g. producing an incorrect result). Most bugs arise due to mistakesand errors made by the programmer, either in the program source code or its design, and a few are caused by compilers producing incorrect code. A program that contains a large number of bugs, and/or bugs that seriously interfere with its functionality, is said to be buggy. Reports detailing bugs in a program are commonly known as bug reports, fault reports or problem reports.

2.1.1 Types of Bugs

1. **Type I error** [39], also known as an *error of the first kind*, an α error, or a *false positive*: the error of rejecting a null hypothesis when it is actually true. It occurs when we are observing a difference when in truth there is none, thus indicating a test of poor specifiction. Type I error can be viewed as the error of *excessive credulity* [46] or even hallucination.

2. **Type II error** [39], also known as an *error of the second kind*, a β error, or a *false negative*: the error of failing to reject a null hypothesis when in fact we should have rejected it. In other words, this is the error of failing to observe a difference when in truth there is one, thus indicating a test of poor sensitivity. Type II error can be viewed as the error of excessive skepticism [47] or myopia.

3. Here we present the bugs those can be branched from the two above bugs and their causes [40]:

   - **Conceptual error**, the causes are: code is syntactically correct, but the programmer or designer intended it to do something else.
   - **Arithmetic bugs**,the causes are: division by zero, arithmetic overflow or underflow, loss of arithmetic precision due to rounding or numerically unstable algorithms.
   - **Logic bugs**, the causes are: infinite loops and infinite recursion, off by one error, counting one too many or too few when looping.
   - **Syntax bugs**, the causes are: use of the wrong operator,
   - **Resource bugs**, the causes are: null pointer dereference, using an uninitialized variable, access violations, resource leaks; where a finite system resource such as memory or file handles are exhausted by repeated allocation without release, buffer overflow, in which a program tries to store data past the end of allocated storage, This may or may not lead to an access violation or storage violation.
   - **Multi-threading programming bugs**, the causes are: deadlock, race condition, concurrency errors in critical sections, mutual exclusions and other features of concurent processing etc.
   - **Interfacing bug**, the causes are: incorrect API usage, incorrect protocol implementation, incorrect hardware handling.
   - **Teamworking bugs**, the causes are: unpropagated updates; e.g. programmer changes "myAdd" but forgets to change "mySubtract", which uses the same algorithm.

2.1.2 Bug Handling

It [40] is common practice for software to be released with known bugs that are considered non-critical, that is, that do not affect most users' main experience with the product. While software products may, by definition, contain any number of unknown bugs, measurements during testing can provide an estimate of the number of likely bugs remaining; this becomes more reliable the longer a product is tested and developed ("if we had 200 bugs last version, we should have 100 this version").

It is often considered impossible to write completely bug-free software of any real complexity. So bugs are categorized by severity, and low-severity non-critical bugs are tolerated, as they do not affect the proper operation of the system for most users.

Like any other part of engineering management, bug management must be conducted carefully and intelligently because "what gets measured gets done" [38] and managing purely by bug counts can have unintended consequences.

2.2 Debugging

Debugging [41] is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware, thus making it behave as expected. Debugging tends to be harder when various subsystems are tightly coupled, as changes in one may cause bugs to emerge in another.

As software and electronic systems have become generally more complex, the various common debugging techniques have expanded with more methods to detect anomalies, assess impact, and schedule software patches or full updates to a system. The words *anomaly* and *discrepancy* can be used, as being more neutral terms, to avoid the words *error* and *defect* or *bug* where there might be an implication that all so-called errors, defects or bugs must be fixed [41].

Debugging ranges, in complexity, from fixing simple errors to performe lengthy and tiresome tasks of data collection, analysis, and scheduling updates. The debugging skill of

the programmer can be a major factor in the ability to debug a problem, but the difficulty of software debugging varies greatly with the complexity of the system, and also depends, to some extent, on the programming language(s) used and the available tools, such as debuggers. Debuggers are software tools which enable the programmer to monitor the execution of a program, stop it, re-start it, set breakpoints, and change values in memory.

Generally [41], high-level programming languages, such as *Java*, make debugging easier, because they have features such as *exception handling* that make real sources of erratic behavior easier to spot. In programming languages such as *C* or *assembly*, bugs may cause silent problems such as *memory corruption*, and it is often difficult to see where the initial problem happened. In those cases, *memory debugger* tools may be needed.

2.3 Embedded Systems

An embedded system [24] is a computer system designed to perform one or a few dedicated functions [25] citehea often with real-time computing constraints. It is embedded as part of a complete device often including hardware and mechanical parts. By contrast, a general-purpose computer, such as a personal computer is designed to be flexible and to meet a wide range of end-user needs. Embedded systems control many devices in common use today [27]. A few characteristics of embedded systems are [24]:

- Embedded systems are designed to do some specific task, rather than be a general-purpose computer for multiple tasks.
- Embedded systems are not always standalone devices, it may consist of small, computerized parts within a larger device that serves a more general purpose.
- The program instructions written for embedded systems are referred to as firmware, and are stored in read-only memory or flash memory chips.

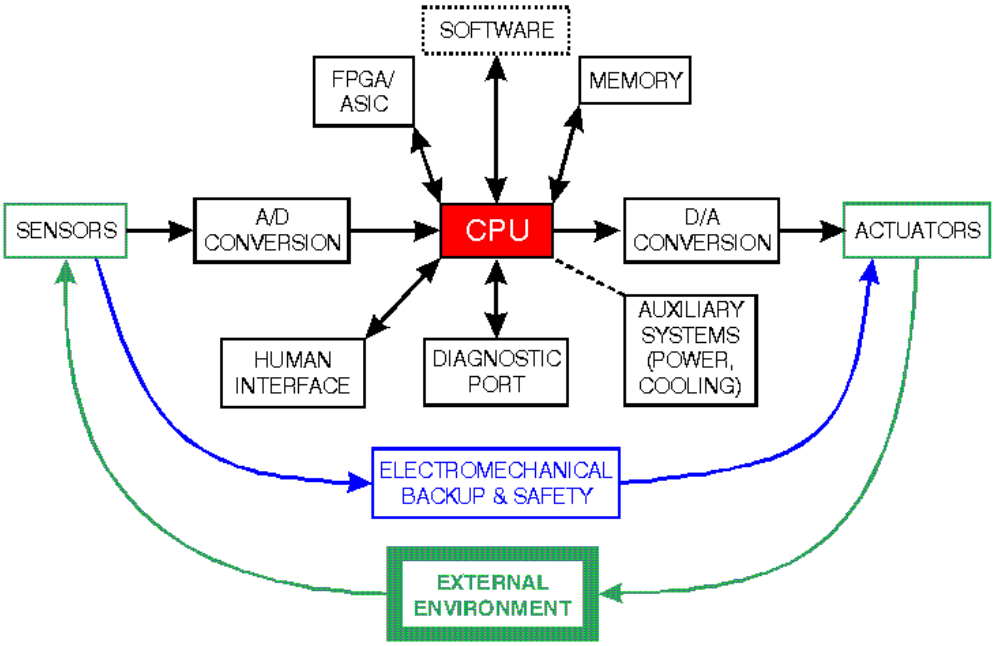

Figure 2.1: An embedded system encompasses the CPU as well as many other resources.

2.3.1 Embedded Processors

Embedded processors can be broken into two broad categories: ordinary microprocessors (μP) and microcontrollers (μC), which have many more peripherals on chip, reducing cost and size. There are Von Neumann as well as various degrees of Harvard architectures, RISC as well as non-RISC and VLIW types of processors are available in market. Few of these are: 65816, 65C02, 68HC08, 68HC11, 68k, 78K0R/78K0, 8051, ARM, AVR, AVR32, Blackfin, C167, Coldfire, COP8, Cortus APS3, eZ8, eZ80, FR-V, H8, HT48, M16C, M32C, MIPS, MSP430, PIC, PowerPC, R8C, RL78, SHARC, SPARC, ST6, SuperH, TLCS-47, TLCS-870, TLCS-900, Tricore, V850, x86, XE8000, Z80, AsAP etc.

2.3.2 Embedded System Feature

The figure 2.1 illustrates the peripheral environment of an embedded processor as well as a construction of a basic embedded system. Embedded systems are very much accustomed

to various need to be guaranteed at the time of development. Few of them are illustrated below [23].

- **Real Time** :Real time system operation means that the correctness of a computation depends, in part, on the time at which it is delivered. In many cases the system design must take into account worst case performance.
- **Small size, low weight** :Many embedded computers are physically located within some larger artifact. Therefore, their form factor may be dictated by aesthetics, form factors existing in pre-electronic versions, or having to fit into interstices among mechanical components.
- **Safe and Reliable** :Some systems have obvious risks associated with failure. In mission-critical applications such as aircraft flight control, severe personal injury or equipment damage could result from a failure of the embedded computer.
- **Harsh environment** :Many embedded systems do not operate in a controlled environment. Excessive heat is often a problem, as in applications involving combustion.
- **Cost sensitivity** :Even though embedded computers have stringent requirements, cost is almost always an issue.
- **Less power requirement** :Low power management is crucial in this case.

2.4 Embedded Software

Embedded software is computer software which plays an integral role in the electronics it is supplied with. Embedded software's principal role is the interaction with the physical world. Embedded software is 'built in' to the electronics in cars, telephones, audio equipment, robots, appliances, toys, security systems, pacemakers, televisions and digital watches, for example. One such open source example of embedded software is BusyBox [22] from Linux distribution.

Embedded Operating System : An embedded operating system is an operating system for embedded computer systems. These operating systems are designed to be compact, efficient, and reliable, forsaking many functions that non-embedded computer operating systems provide, and which may not be used by the specialized applications they run.

2.4.1 Characteristics of Embedded Software

An arrogant view of embedded software is that it is just software on small computers. This view is naive. Timeliness, concurrency, liveness, reactivity, and heterogeneity need to be an integral part of the programming abstractions. They are essential to the correctness of a program [37].

1. **Timeliness** :Time has been systematically removed from theories of computation. "Pure" computation does not take time, and has nothing to do with time.

2. **Concurrency** :Embedded systems rarely interact with only a single physical process. They must simultaneously react to stimulus from a network and from a variety of sensors, and at the same time, retain timely control over actuators. This implies that embedded software is concurrent.

3. **Interfaces** :Software engineering has experienced major improvements over the last decade or so through the widespread use of object oriented design which is a component technology, in the sense that a large complicated design is composed of pieces that expose interfaces that abstract their own complexity.

4. **Heterogeneity** :Heterogeneity is an intrinsic part of computation in embedded systems. It mixes computational styles and implementation technologies.

5. **Reactivity** :Reactive systems are those that react continuously to their environment at the speed of the environment. Harel and Pnueli [33] and Berry [34] contrast them with interactive systems, which react with the environment at their own speed, and

transformational systems, which simply take a body of input data and transform it into a body of output data.

2.4.2 Embedded Software Architecture

Embedded software contains some feature of important measure, such as [24]:

- **Simple control loop** :In this design, the software simply has a loop. The loop calls subroutines, each of which manages a part of the hardware or software.
- **Interrupt controlled system** :Some embedded systems are predominantly interrupt controlled. This means that tasks performed by the system are triggered by different kinds of events.
- **Cooperative multitasking** :A nonpreemptive multitasking system is very similar to the simple control loop scheme, except that the loop is hidden in an API.
- **Preemptive multitasking** :In this type of system, a low-level piece of code switches between tasks or threads based on a timer (connected to an interrupt).
- **Micro and Exo-kernel** :A microkernel is a logical step up from a real-time OS. The usual arrangement is that the operating system kernel allocates memory and switches the CPU to different threads of execution. In general, microkernels succeed when the task switching and intertask communication is fast, and fail when they are slow. Exokernels communicate efficiently by normal subroutine calls.
- **Monolithic kernels** :In this case, a relatively large kernel with sophisticated capabilities is adapted to suit an embedded environment.
- **Exotic custom operating systems** :A small fraction of embedded systems require safe, timely, reliable or efficient behavior unobtainable with the one of the above architectures.

2.4.3 Models of Computation

There are many models of computation, each dealing with concurrency and time in different ways. Here we outline some of the most useful models for embedded software [37].

1. **Dataflow** : In dataflow models, actors are atomic (indivisible) computations that are triggered by the availability of input data. Connections between actors represent the flow of data from a producer actor to a consumer actor.

2. **Time Triggered** :Some systems with timed events are driven by clocks, which are signals with events that are repeated indefinitely with a fixed period.

3. **Synchronous/Reactive** :In the synchronous/reactive (SR) model of computation, connections between components represent data values that are aligned with global clock ticks, as with time-triggered approaches [37].

4. **Rendezvous** :In synchronous message passing, the components are processes, and processes communicate in atomic, instantaneous actions called rendezvous. If two processes are to communicate, and one reaches the point first at which it is ready to communicate, then it stalls until the other process is ready to communicate.

5. **Finite State Machines** :All [37] of the models of computation considered so far are concurrent. It is often useful to combine these concurrent models hierarchically with finite-state machines (FSMs) to get modal models. FSMs are different from any of the models we have considered so far in that they are strictly sequential. A component in this model is called a state or mode, and exactly one state is active at a time. The connections between states represent transitions, or transfer of control between states. Execution is a strictly ordered sequence of state transitions. Transition systems are a more general version, in that a given component may represent more than one system state (and there may be an infinite number of components).

2.5 BusyBox

In this thesis we have experimented on BusyBox [22], a de-facto standard for Linux in emebedded systems. BusyBox is a fairly comprehensive set of programs needed to run a Linux system. Moreover it is the de-facto standard for Embedded Linux systems, providing many standard Linux utilities, but having small code size (size of executables), than the GNU Core Utilities. BusyBox provides compact replacements for many traditional full-blown utilities found on most desktop and embedded Linux distributions. Examples include the le utilities such as ls, cat, cp, dir, head, and tail. BusyBox also provides support for more complex operations, such as ifcong, netstat, route, and other network utilities. BusyBox is remarkably easy to congure, compile, and use, and it has the potential to signicantly reduce the overall system resources required to support a wide collection of common Linux utilities. BusyBox in general case can be built on any architecture supported by gcc.

BusyBox is modular and highly configurable, and can be tailored to suit particular requirements. The package includes a configuration utility similar to that used to congure the Linux kernel. The commands in BusyBox are generally simpler implementations than their full-blown counterparts. In some cases, only a subset of the usual command line options is supported. In practice, however, the BusyBox subset of command functionality is more than sufcient for most general embedded requirements.

The BusyBox bundle functions as a single executable where the different utilities are actually passed on at the command line for separate invocation. It is not possible to build the individual utilities separately and run them stand alone. For example, for running the arp utility, we need to invoke BusyBox as busybox arp-Ainet and record the execution trace. Since we work on the binary level, the buggy implementation for us is the BusyBox binary, which has a large code base (about 121000 lines of code).

2.5.1 Overall View of BusyBox

BusyBox contains seveal standrad Linux utilities in it. Figure 2.2 shows that. Several Projects and products that do use BusyBox are given below in figures 2.3 and 2.4 respectively [22].

2.5.2 Usage

BusyBox is a multi-call binary [22]. A multi-call binary is an executable program that performs the same job as more than one utility program. That means there is just a single BusyBox binary, but that single binary acts like a large number of utilities. This allows BusyBox to be smaller since all the built-in utility programs (applets) can share code for many common operations. We can also invoke BusyBox by issuing a command as an argument on the command line. For example, entering
/bin/busybox ls
will also cause BusyBox to behave as 'ls'.

2.5.3 Bugs in BusyBox

KLEE [17] has reported some bugs in BusyBox 1.4.2. by a test generation method. Klee had checked all 279 BusyBox tools in series to demonstrate its applicability to bug finding. It has been seen that 21 bugs are present in BusyBox. We tried them on BusyBox version 1.4.2 and found (version 1.16.0) 6 of them persists namely, arp -Ainet, tr [, top d, printf %Lu, ls -co, install -m.

[, [[, acpid, addgroup, adduser, adjtimex, ar, arp, arping, ash, awk, basename, beep, blkid, brctl, bunzip2, bzcat, bzip2, cal, cat, catv, chat, chattr, chgrp, chmod, chown, chpasswd, chpst, chroot, chrt, chvt, cksum, clear, cmp, comm, cp, cpio, crond, crontab, cryptpw, cttyhack, cut, date, dc, dd, deallocvt, delgroup, deluser, depmod, devmem, df, dhcprelay, diff, dirname, dmesg, dnsd, dnsdomainname, dos2unix, du, dumpkmap, dumpleases, echo, ed, egrep, eject, env, envdir, envuidgid, ether-wake, expand, expr, fakeidentd, false, fbset, fbsplash, fdflush, fdformat, fdisk, fgrep, find, findfs, flash_eraseall, flash_lock, halt, hd, hdparm, head, hexdump, hostid, hostname, httpd, hush, hwclock, id, ifconfig, ifdown, ifenslave, ifplugd, ifup, inetd, init, inotifyd, logread, losetup, lpd, lpq, lpr, ls, lsattr, lsmod, lspci, lsusb, lzmacat, lzop, lzopcat, makedevs, makemime, man, md5sum, mdev, patch, pgrep, pidof, ping, ping6, pipe_progress, pivot_root, pkill, popmaildir, poweroff, printenv, printf, ps, pscan, pwd, raidautorun, rdate, rdev, readahead, readlink, readprofile, realpath, reboot, reformime, renice, reset, resize, rm, rmdir, rmmod, route, rpm, rpm2cpio, rtcwake, run-parts, runlevel, runsv, runsvdir, rx, script, scriptreplay, sed, sendmail, seq, setarch, setconsole, setfont, setkeycodes, setlogcons, setsid, setuidgid, sh, sha1sum, sha256sum, sha512sum, showkey, slattach, sleep, softlimit, sort, split, ostart-stop-daemon, stat, strings, stty, su, sulogin, sum, sv, svlogd, swapoff, swapon, switch_root, sync, sysctl, syslogd, tac, tail, tar, taskset, tcpsvd, tee, telnet, telnetd, test, tftp, tftpd, time, uudecode, uuencode, vconfig, vi, vlock, volname, wall, watch, watchdog, wc, wget, which, who, whoami, xargs, yes, zcat, zcip, msh, mt, mv etc.

Figure 2.2: The commands incorporated in BusyBox version-1.16.0.

```
* buildroot--A configurable means for building users' own busybox/
  uClibc based system systems, maintained by the uClibc developers.
* OpenWrt--A Linux distribution for embedded devices, based on
  buildroot.
* Tiny Core Linux--A very small minimal Linux GUI Desktop.
* PTXdist--Another configurable means for building your own busybox
  based systems.
* Debian installer (boot floppies) project.
* Red Hat installer.
* Slackware Installer.
* Gentoo Linux install/boot CDs.
* The Mandriva installer.
* Linux Embedded Appliance Firewall--The sucessor of the Linux Router
  Project, supporting all sorts of embedded Linux gateways, routers,
  wireless routers, and firewalls.
* Build Your Linux Disk.
* AdTran - VPN/firewall VPN Linux Distribution.
* mkCDrec - make CD-ROM recovery.
* Partition Image.
* Familiar Linux--A linux distribution for handheld computers.
* Netstation.
* GNU/Fiwix Operating System.
* TimeSys real-time Linux.
* MoviX--Boots from CD and automatically plays every video file on the
  CD.
* Salvare--More Linux than tomsrtbt but less than Knoppix, aims to
  provide a useful workstation as well as a rescue disk and many more.
```

Figure 2.3: List of projects that use the BusyBox.

```
* Galaxy MGB4500 Raid Pro NAS.
* StoryBox Ultimate uses BusyBox v1.1.3.
* Western Digital's ShareSpace network attached storage device.
* RD129 embedded board from ELPA.
* EMTEC MovieCube R700 uses BusyBox 1.1.3.
* The Kerbango Internet Radio.
* LinuxMagic VPN Firewall.
* Cyclades-TS and other Cyclades products.
* Linksys WRT54G - Wireless-G Broadband Router.
* Dell TrueMobile 1184.
* NetGear WG602 wireless router with sources here.
* ASUS WL-300g Wireless LAN Access Point with source here.
* Belkin 54g Wireless DSL/Cable Gateway Router with source here.
* Acronis PartitionExpert 2003.
* U.S. Robotics Sureconnect 4-port ADSL router with source here.
* ActionTec GT701-WG Wireless Gateway/DSL Modem with source here.
* DLink--Model GSL-G604T, DSL-300T.
* Siemens SE515 DSL router.
* Free Remote Windows Terminal.
* ZyXEL Routers
  etc.
```

Figure 2.4: List of products that use the BusyBox.

Chapter 3

Related Work

In this chapter we provide a brief illustration to the various research works performed on debugging aspects of embedded software.

[20] implemented a delta debugging algorithm which isolates the relevant variables and values by systematically narrowing the state difference between a passing run and a failing run–by assessing the outcome of altered executions to determine whether a change in the program state makes a difference in the test outcome. Applying Delta Debugging to multiple states of the program automatically reveals the cause-effect chain of the failure e.g., the variables and values that cause the failure. In [21] delta debugging applies the scientic method of debugging to narrow down the set of failure-inducing circumstances automatically–circumstances such as the program input, changes to the program code, or executed statements.

[19] presents an approach for identifying failure-inducing changes in a system with an associated regression test suite. In contrast to extreme programming (XP) where the number of changes between test runs tends to be small, where it is assumed that the size of an edit can become sufciently large to make the identication of failure-inducing changes a difcult task, and to make automated assistance with this task desirable. The paper [18] generates test cases that make the effect of software changes observable through difference in program outputs. A test case for regression testing should drive the changed program to execute the changes and the program states affected by the changes should result in difference in program outputs. It uses symbolic execution on program traces to guide the exploration of program paths.

Klee [17] is a new symbolic execution tool capable of automatically generating tests that

achieve high coverage on a diverse set of complex and environmentally-intensive programs. It uses a very important Unix utility suite BusyBox to test. [16] investigates the possibility of using the golden implementation as a reference model in software debugging. They performed dynamic slicing as well as weakest precondition computation of the observable error with respect to the statement in the dynamic slice.

[14] proposed a control flow based difference metric for this purpose. The difference metric takes into account the sequence of statement instances (and not just the set of these instances) executed in the two runs, by locating branch instances with similar contexts but different outcomes in the failing and the successful runs. Given a failing run π_f and a pool of successful runs S , we choose the successful run π_s from S whose execution trace is closest to π_f in terms of the difference metric. A bug report is then generated by returning the difference between π_f and π_s . A method [13] constructs a successful program run by toggling the outcomes of some conditional branch instances in the failing run. If such a successful run exists, pro- gram statements for these branches are returned as bug report. Here is an issue to generate or choose a "suitable" successful run(a run which does not demonstrate error).

[12] described a tool called IODINE (Implementation of Dynamic Invariant Extraction) to automatically extract likely properties based on design simulations. This approach extracts dynamic invariants by hypothesizing a set of invariants across one or more variables in the design and analyzing the behavior of the design over a set of inputs. As potential invariants are falsified during program runs, they are removed from the candidate set, leaving only invariants which are true for the given input set. In [11] spectrum-based fault localization (SFL) technique is used which shortens the test-diagnose-repair cycle by reducing the debugging effort. As a light-weight automated diagnosis technique it can easily be integrated with existing testing schemes. Since SFL is based on discovering statistical coincidences between system failures and the activity of the different parts of a system, its diagnostic accuracy is inherently limited.

[10] integrates the potential of delta debugging algorithm with the benefit of forward

and backward dynamic program slicing to narrow down the search for faulty code. The approach uses delta debugging algorithm to identify a minimal failure-inducing input and uses this input to compute a forward dynamic slice and then intersect the statements in this forward dynamic slice with the statements in the backward dynamic slice of the erroneous output to compute a failure-inducing chop.

[9] addressed the problem pertaining to the reveal of source values for the globally optimized behavioral specifications. It presented a design-for-debugging approach for a symbolic debugger to retrieve and display the value of a variable correctly and efficiently in response to a user inquiry about the variable in the source specification.

[7] described the design principle to classify a single utility function that integrates the various debugging heuristics. It presented a strategy for automatically debugging programs given sampled data from thousands of actual user runs. The goal was to pinpoint those features that were most correlated with cashes, accomplished by maximizing an appropriately defined utility function. Nicholas et. al. [6] described the implementation of shadow memory in Memcheck, a popular memory checker built with Valgrind, a dynamic binary instrumentation framework. Mutation testing [5] measured the adequacy of a test suite by seeding artificial defects (mutations) into a program. If a mutation is not detected by the test suite, it usually mean that the mutation kept the program's semantics unchanged and can not be detected by any test. JAVALANCHE framework found the mutations that violated invariants were significantly likely to be detectable by a test suite.

Chapter 4

Memory Error Detection

In this chapter we will present a methodology to localize memory related errors. We will employ Valgrind to implement this technique. Our methodology will work on two bugs of BusyBox namely `arp` and `top`.

4.1 Valgrind

4.1.1 Introduction

The Valgrind tool is the well known memory error detector extensively used in various research work and corporate sector. The Valgrind tool suite provides a number of debugging and profiling tools that help to make the programs faster and more correct. The most popular of these tools is called Memcheck. It can detect many memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behavior. Valgrind requires a program compiled with -g so that Memchecks error messages can include exact line numbers. We run this tool in command line by providing

valgrind –leak-check=yes myprog arg1 arg2

where *myprog* is a pre-compiled program binary, *arg1* and *arg2* are the arguments passed along with the program. The *–leak-check* option turns on the detailed memory leak detector. Here *Memcheck* is the default tool. *Memcheck* will issue messages about memory errors and leaks that it detects.

Valgrind comes with a set of tools each of which performs some kind of debugging, profiling, or similar task that helps to improve programs. Some of them are as below.

- The default tool that comes along with Valgrind, is *Memcheck*, a memory error detector.
- *Cachegrind* – a cache and branch-prediction pro-filer.
- *Massif* – a heap profiler.
- *Helgrind* – a thread error detector.
- *Memory leak detector.*
- *Conditional jump or uninitialized value dependent move detector.*
- *Invalid Read / Write Detector.*

4.1.2 Interpreting Memchecks Output

Here is an example C program, in a file called a.c, with a memory error and a memory leak.
Most error messages look like the following, which describes problem 1 from 4.1, the heap block overrun: Things to notice from 4.2 are as below.

- There is a lot of information in each error message.
- The 19182 is the process ID; it is usually unimportant.
- The first line ("Invalid write...") tells what kind of error it is. Here, the program wrote to some memory it should not have due to a heap block overrun.
- Below the first line is a stack trace telling, where the problem occurred. Stack traces can get quite large, and be confusing, especially if we are using the C++ STL. If the stack trace is not big enough, using the *–num-callers* option will make it bigger.

```
#include <stdlib.h>
void f(void)
{
  int* x = malloc(10 * sizeof(int));
  x[10] = 0;    // problem 1: heap block overrun
}               // problem 2: memory leak -- x not freed
int main(void)
{
  f();
  return 0;
}
```

Figure 4.1: Example of a C program having memory leak error.

```
==19182== Invalid write of size 4
==19182==   at 0x804838F: f (example.c:6)
==19182==   by 0x80483AB: main (example.c:11)
==19182== Address 0x1BA45050 is 0 bytes after a block of size 40 allocd
==19182==   at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==   by 0x8048385: f (example.c:5)
==19182==   by 0x80483AB: main (example.c:11)
```

Figure 4.2: Output of 4.1 run by Valgrind tool.

```
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (a.c:5)
==19182==    by 0x80483AB: main (a.c:11)
```

Figure 4.3: Memory leak message part of output of 4.1 run by Valgrind tool.

- The code addresses (eg. 0x804838F) are usually unimportant, but occasionally crucial for tracking down weirder bugs.
- Some error messages have a second component which describes the memory address involved. This one shows that the written memory is just past the end of a block allocated with malloc() on line 5 of example.c.

One thing that is to be noted is, it is worth fixing errors in the order they are reported, as later errors can be caused by earlier errors. Failing to do this is a common cause of difficulty with Memcheck. Memory leak messages look like this:

The stack trace tells us where the leaked memory was allocated. Memcheck cannot tell why the memory leaked, unfortunately. (Ignore the "vg_replace_malloc.c", that is an implementation detail.) There are several kinds of leaks; the two most important categories are:

- "definitely lost": the program is leaking memory, should be fixed.
- "probably lost": the program is leaking memory, unless programmer is doing funny things with pointers (such as moving them to point to the middle of a heap block).

Some other kind of options and errors are also get detected by Valgrind, but are out of scope of this writing.

4.2 Bug Analysis

4.2.1 Analyzing the Arp Bug in BusyBox

The `arp` utility manages the kernel's network neighbor cache. It can add or delete entries to the cache, or display the cache's current content. There is a bug in the BusyBox `arp` implementation: running `arp` with the command-line option **-Ainet** results in a *segmentation fault.*

To isolate the root cause of this error, we ran the arp utility of BusyBox through Valgrind using the argument -Ainet. This results in segmentation fault when run in BusyBox version 1.4.2, on valgrind. Figure 4.4 shows the test results.

mc_replace_strmem.c, *applets.c*, *busybox.c* are few of the implementation details shown in 4.4. These details are usually unimportant in respect to our methodology. Figure 4.5 shows a fragment of the source code of `arp` in BusyBox. With the command-line argument `-Ainet`, line 454 sets the `ARP_OPT_A` mask in the variable `option_mask32`. Because no `H` or `t` option was given in the command line, `hw_type` was set to NULL. The bug is at line 462: instead of checking the mask of hardware type, the program checks for the mask of address family, `ARP_OPT_A`, which led to line 463, which passed the NULL `hw_type` into `get_hwtype` function, and caused a segmentation fault at line 936 due to a NULL argument in the string comparison function `strcmp`.

4.2.2 Analyzing the Top Bug in BusyBox

The top program provides a dynamic real-time view of a running system. It can display system summary information as well as a list of tasks currently being managed by the Linux kernel. The types of system summary information shown and the types, order and size of information displayed for tasks are all user configurable and that configuration can be made persistent across restarts.

Top which is run with option d is buggy in busybox v1.4.2. If we run BusyBox top

```
==1571== Use of uninitialized value of size 8
==1571==    at 0x4A06794: strcmp (mc_replace_strmem.c:341)
==1571==    by 0x44D7BD: get_hwtype (interface.c:936)
==1571==    by 0x444B77: arp_main (arp.c:463)
==1571==    by 0x407C08: run_applet_by_name (applets.c:489)
==1571==    by 0x407DDC: busybox_main (busybox.c:143)
==1571==    by 0x407AB7: run_applet_by_name (applets.c:480)
==1571==    by 0x407E56: main (busybox.c:72)
==1571==
==1571== Invalid read of size 1
==1571==    at 0x4A06794: strcmp (mc_replace_strmem.c:341)
==1571==    by 0x44D7BD: get_hwtype (interface.c:936)
==1571==    by 0x444B77: arp_main (arp.c:463)
==1571==    by 0x407C08: run_applet_by_name (applets.c:489)
==1571==    by 0x407DDC: busybox_main (busybox.c:143)
```

Figure 4.4: BusyBox Arp run with valgrind.

```
930 const struct hwtype *get_hwtype (const char *name)
931 {
932    const struct hwtype * const *hwp;
933    hwp = hwtypes;
934    while (*hwp != NULL)
935    {
936      if (!strcmp((*hwp)->name, name))
937            return (*hwp);
938        hwp++;
939    }
940   return NULL;
941 }

444 int arp_main(int argc, char **argv)
445 {
446     char *hw_type;
447     char *protocol;
448
449     /* Initialize variables... */
450     ap = get_aftype(DFLT_AF);
451     if (!ap)
452         bb_error_msg_and_die("%s: %s not supported", DFLT_AF, "address
            family");
453
454     getopt32(argc, argv, "A:p:H:t:i:adnDsv", &protocol, &protocol,
455                   &hw_type, &hw_type, &device);
456     argv += optind;
457     if (option_mask32 & ARP_OPT_A || option_mask32 & ARP_OPT_p) {
```

```
458         ap = get_aftype(protocol);
459         if (ap == NULL)
460             bb_error_msg_and_die("%s: unknown %s",protocol, "address
                family");
461     }
462     if (option_mask32 & ARP_OPT_A || option_mask32 & ARP_OPT_p) {
463         hw = get_hwtype(hw_type);
464         if (hw == NULL)
465             bb_error_msg_and_die("%s: unknown %s", hw_type, "hardware
                type");
466         hw_set = 1;
467     }
468     //if (option_mask32 & ARP_OPT_i)... -i
469
470     if (ap->af != AF_INET) {
471         bb_error_msg_and_die("%s: kernel only supports 'inet'", ap->name );
472     }
473
474     /* If no hw type specified get default */
475     if (!hw) {
476         hw = get_hwtype(DFLT_HW);
477         if (!hw)
478             bb_error_msg_and_die("%s: %s not
                supported",DFLT_HW, "hardware type");
479     }
       ....
    }
```

Figure 4.5: BusyBox arp application illustrating *get_hytype()* and *arp_main()*.

```
==1575== Invalid read of size 1
==1575==    at 0x43310E: xstrtou_range_sfx (xatonum_template.c:27)
==1575==    by 0x463605: top_main (top.c:431)
==1575==    by 0x407C08: run_applet_by_name (applets.c:489)
==1575==    by 0x407DDC: busybox_main (busybox.c:143)
==1575==    by 0x407AB7: run_applet_by_name (applets.c:480)
==1575==    by 0x407E56: main (busybox.c:72)
==1575==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==1575==    Process terminating with default action of signal 11 (SIGSEGV)
==1575==    Access not within mapped region at address 0x0
==1575==    at 0x43310E: xstrtou_range_sfx (xatonum_template.c:27)
==1575==    by 0x463605: top_main (top.c:431)
==1575==    by 0x407C08: run_applet_by_name (applets.c:489)
==1575==    by 0x407DDC: busybox_main (busybox.c:143)
==1575==    by 0x407AB7: run_applet_by_name (applets.c:480)
==1575==    by 0x407E56: main (busybox.c:72)
```

Figure 4.6: BusyBox top run with valgrind.

with option d, it will crash and a segmentation fault is reported. Option d in top must be followed by argument which specifies the length of the refresh delay in the process statistic (in seconds). The top utility should behave similarly when invoked with *-d* or *d*. The – is complementary in this command. In the normal execution of top on GNU, both *top d* and *top -d* wait for the parameter for refresh delay. However, BusyBox top behaves differently. Figure 4.6 shows the top results.

Figure 4.7 shows a fragment of the source code of `top` in BusyBox. With the command-line argument `d`, there is a crash in line 431 as reported by Valgrind. BusyBox's top uses a native getopt32 and checks for '-' which should have been ignored by top program. Going

```
414 int top_main(int argc, char **argv)
415 {
416     int count, lines, col;
417     unsigned interval = 5;
        /* default update rate is 5 seconds */
418     unsigned iterations = UINT_MAX;
        /* 2^32 iterations by default :) */
419     char *sinterval, *siterations;
420 #if ENABLE_FEATURE_USE_TERMIOS
421     struct termios new_settings;
422     struct timeval tv;
423     fd_set readfds;
424     unsigned char c;
425 #endif /* FEATURE_USE_TERMIOS */
426
427     /* do normal option parsing */
428     interval = 5;
429     opt_complementary = "-";
430     getopt32(argc, argv, "d:n:b", &sinterval, &siterations);
431     if (option_mask32 & 0x1) interval = xatou(sinterval);// -d
432     if (option_mask32 & 0x2) iterations = xato(siterations);// -n
433     //if (option_mask32 & 0x4) // -b
434
435     /* change to /proc */
436     xchdir("/proc");
437 #if ENABLE_FEATURE_USE_TERMIOS
438     tcgetattr(0, (void *) &initial_settings);
439     memcpy(&new_settings, &initial_settings, sizeof(struct termios));
```

```
440     /* unbuffered input, turn off echo */
441     new_settings.c_lflag &= ~(ISIG | ICANON | ECHO | ECHONL);
442
443     signal(SIGTERM, sig_catcher);
444     signal(SIGINT, sig_catcher);
445     tcsetattr(0, TCSANOW, (void *) &new_settings);
446     atexit(reset_term);
447 #endif /* FEATURE_USE_TERMIOS */
448
449 #if ENABLE_FEATURE_TOP_CPU_USAGE_PERCENTAGE
450     sort_function[0] = pcpu_sort;
451     sort_function[1] = mem_sort;
452     sort_function[2] = time_sort;
453 #else
454     sort_function = mem_sort;
455 #endif /* FEATURE_TOP_CPU_USAGE_PERCENTAGE */
456
457     while (1) {
458         procps_status_t *p = NULL;
```

Figure 4.7: BusyBox top application.

```
....
if (spec_flgs & ALL_ARGV_IS_OPTS) {
   /* process argv is option, for example "ps" applet */
   if (pargv == NULL)
      pargv = argv + optind;
   while (*pargv) {
      // printf ("argv non option: %s\n", *pargv);
      c = **pargv;
      if (c == '\0') {
          pargv++;
      }
      else {
          (*pargv)++;
          goto loop_arg_is_opt;
      }
   }
}
....
```

Figure 4.8: BusyBox getopt32.

inside the getopt32 function in Line 430, we find the code snippet shown in Figure 4.8.

The pargv here is not NULL, thus there are some non-options to be processed. In our case here, the non-options to be processed is string 'd' and the following while-loop treat it as an option. However as soon as 'd' is treated as an option, it is no longer checked whether 'd' requires an argument. Thus the argument stored in sinterval in line 431 of top_main is NULL leading to the crash.

Chapter 5

Invariant Analysis

In this chapter we will present a debugging methodology incorporating invariant analysis. This methodology works on a block of code from BusyBox and illustrates the bug tracking concept.

5.1 Introduction

5.1.1 Invariant

An invariant is a property that holds at a certain point or points in a program; these are often seen in assert statements, documentation, and formal specifications. Invariants can be useful in program understanding and a host of other applications. Examples include"field > abs(y)";"y = 2*x+3";"array a is sorted";"for all list objects lst, lst.next.prev = lst";"for all treenode objects n, n.left.value < n.right.value";"p != null => p.content in myArray"; and many more.

5.1.2 Program Point

Invariants can be checked at arbitrary locations in a program. Two examples are procedure entries and exits, resulting in invariants that correspond to preconditions and post conditions. It can be useful to compute invariants at each procedure exit (return statement) and also to compute an aggregate exit point (as viewed by a client) by generalizing over the individual exit points. object or class invariants are also computed at an aggregate program point (object point), by generalizing over all objects that are observed at entry to

and exit from public methods of a class, that are passed into or returned from methods of other classes, or that are stored in object fields [4].

5.1.3 Nonsensical

Some trace variables and derived variables may represent meaningless expressions; in such a circumstance, the value is said to be nonsensical. That variable appears in the .dtrace file, but its value is marked as nonsensical. Some trace variables and derived variables may not have a value because the expression that computes it cannot be evaluated. Even if implication is not verified between two invariant sets after examining the preconditions, continue to check the implication involving post-conditions. This is somewhat dangerous, in that if the implication does not hold between the preconditions, the invariant sets may be inconsistent, in which case reasoning about the post-conditions is formally nonsensical. Examples include: x when x is uninitialized or de-allocated, x.y when x is null (uninitialized or de-allocated), a[i] when i is outside the bounds of a (uninitialized or de-allocated, or a is null).

5.2 Daikon

5.2.1 Introduction

Daikon is an implementation of dynamic detection of likely invariants; e.g., the Daikon invariant detector reports likely program invariants. Dynamic invariant detection runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. Daikon can detect properties in C, C++, Eiffel, Java, and Perl programs; in spreadsheet files; and in other data sources. Dynamic invariant detection is a machine learning technique that can be applied to arbitrary data. Daikon runs on JVM. It supports several front-end tools such as Kvasir, Mangel-Wurzel, Chicory for diffrent languages to create data trace files. Daikon itself used to detect invariants.

5.2.2 Executing Daikon

To detect invariants for a program with Kvasir, we need to perform two basic tasks: first run the program under Kvasir to create a data trace file (step 1), and then run Daikon over the data trace file to produce invariants (step 2-3). The following instructions are for the wordplay example, which is a C program for demonstration purpose.

1. We compile the program with DWARF-2 debugging information enabled (and all optimizations disabled), such as:
   gcc -gdwarf-2 wordplay.c -o wordplay

2. Then we run the program prepending kvasir-dtrace to the command line, such as:
   kvasir-dtrace ./wordplay

   Kvasir executes additional checks and collects trace information. Kvasir creates a directory named daikon-output under the current directory, and creates the wordplay.dtrace file, which lists both variable declarations and values.

3. We run Daikon on the trace files such as:
   java daikon.Daikon −−config_option daikon.derive.Derivation.disable_derived_variables=true daikon-output/wordplay.dtrace

   The invariants are printed to standard output, and a binary representation of the invariants is written to ‘wordplay.inv.gz.

4. Examine the invariants.

   - Examine the output from running Daikon.
   - Use the PrintInvariants program to display the invariants.

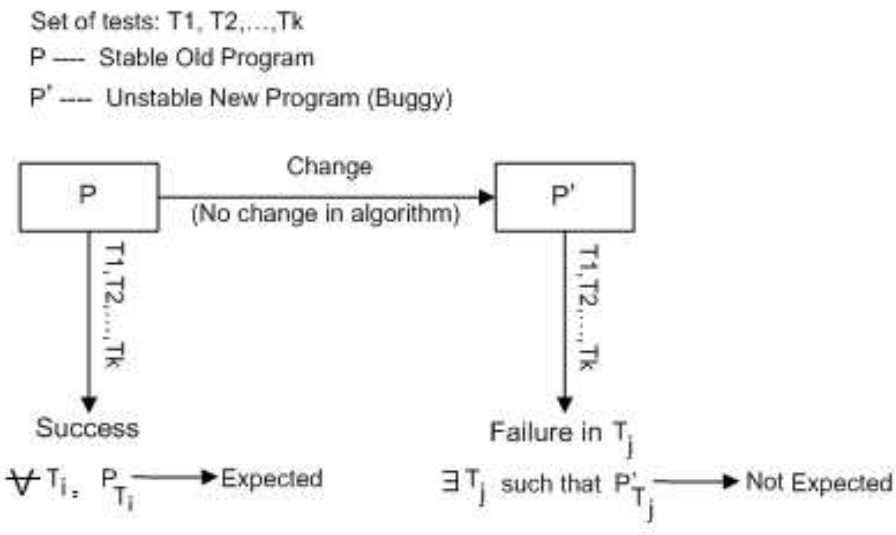

Figure 5.1: Test input feeding into stable and buggy program.

5.3 Methodology

Embedded programs are developed to perform a predefined and specific task. In this respect it should be pointed that with the rapid development of embedded programming structure and huge market requirements, the designers often practice to compress down (lower in size and smaller in complexity) the existing (stable) version of program into newer one. This compels the programmer to change in some parts of the stable code leading to an obvious condition of bug intrusion into the resulting code at development phase keeping overall algorithm fixed, though. Our research objective is to search out the root cause of the bugs present in the newly generated code or buggy code. Figure 5.1 presents the concept.

The figure 5.1 demonstrates the set of tests T_1,T_2,. . .,T_k being fed into the stable program P and buggy program P'. All the test inputs run correctly producing expected result on P, whereas all but T_j show expected output when run by P'. We have to find out the reason of the failure of test input T_j in P'. The figure 5.2 illustrates the same.

In figure 5.3 we have shown our detail methodology to debug an embedded program in-

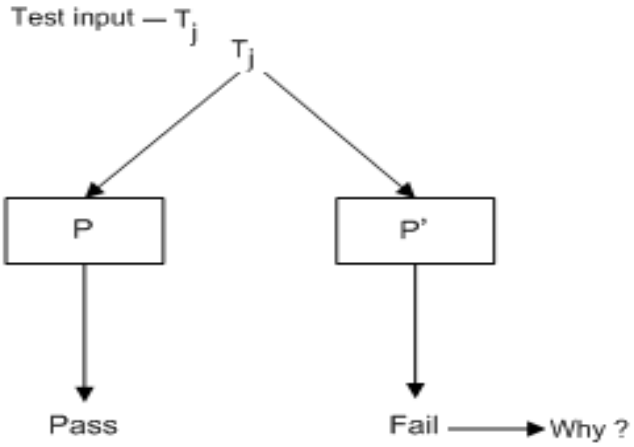

Figure 5.2: Test input T_j passes in P but fails in P'.

corporating invariant analysis. As the diagram illustrates, the test input is fed to both the binaries (debug build executables) of stable and buggy program. Where the given test input passes (successful run) in the execution of P but fails (failed run) in P'. The mentioned successful run and failed run, are meant that the test case produced the expected behavior and unexpected behavior, respectively. This whole thing may lead to correct output, segmentation fault, or program crash, anything to the programmer and the programming measure which truly depends upon the algorithm of the code under investigation.

The next phase of the methodology relies heavily on Daikon. Where we feed the two binaries in the front-end (instrumenter) (kvasir-dtrace for program written in C/ C++) that in turn produces two different dtrace (declaration and values of the variables of code) files to be processed in next stage.

The *dtrace* files are separately fed into the invariant generator to generate *inv* files containing all the invariants of the code. At the last phase of this methodology the *inv* files are compared by invariant comparator, which results an output file containing the differences of the invariants, as a bug report.

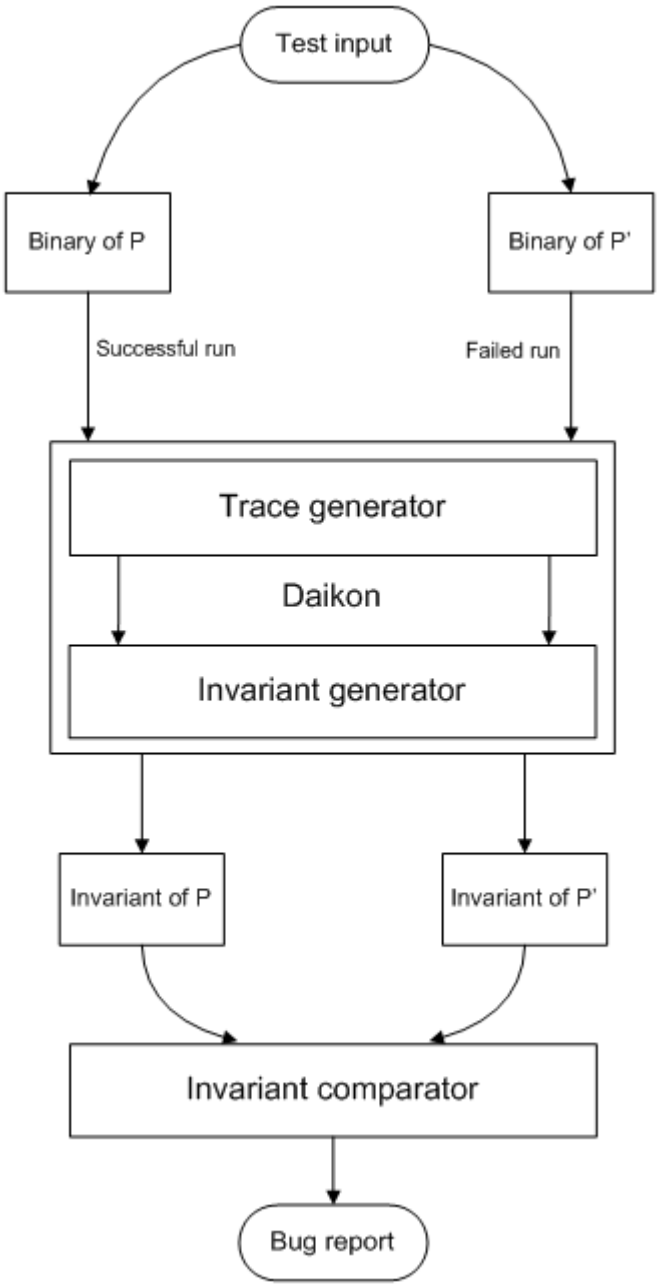

Figure 5.3: Overview of invariant analysis incorporated embedded program debugging approach.

Hence after, we proceed with the bug report to analyze the invariants that were not present in any of the two programs or have changed its value during change of code. This lets to point out the root cause of the bug associated with the code fragment, related to the culprit invariants in the code.

5.4 Concept of Arp Bug Behind Our Approach

In 2.5.3 we introduced the bug named `arp -Ainet` reported in BusyBox by [17]. Running arp with the command-line option -Ainet results in a segmentation fault. `arp Ainet` in command line argument executes well when run in GNU environment, but produces wrong output when run in BusyBox1.4.2. At this point of view we can assume the source program (arp.c) of the above said bug– arp in GNU environment to be stable program P and the same of the BusyBox environment, to be change induced buggy implantation P'. As because the file arp.c is larger in size and complexity, so we have chosen the pseudo format of arp.c from [16] to implement our methodology in simpler way. We have termed it P, whereas the code snippet that we have developed from P (changed) is P'. Both the programs P and P' act as the stable and buggy respectively. Figure 5.4and 5.5 presents P and P' respectively.

5.5 Bug Analysis

Consider an execution of the two programs as *cuArp A inet* and *bbArp A inet*, where bbArp and cuArp are respectively the names of the executables resulting out of P and P'. Evidently, the output of cuArp is as expected inet DFLT_HW, while the output of bbArp is inet NULL (since argv[3] is NULL) which is undesirable. As an objective of investigating the incorrect value of hardware type, we set out to find the root cause as to why the variable hw is set to a NULL value at the end of the program execution [16].

Program P: Old stable inplementation

```
1 #include<stdio.h>
2
3 const char* get_hwtype (const char *name){
4         return name;
5 }
6
7 int main(int argc, char **argv){
8         const char *hw = NULL;
9         const char *ap = NULL;
10        int hw_set = 0;
11        switch (*argv[1]){
12                case 'A': case 'p':{
13                        ap = argv[2];
14                        break;
15                }
16                case 'H': case 't':{
17                        hw = get_hwtype(argv[3]);
18                        hw_set=1;
19                        break;
20                }
21                default: break;
22        }
23        if((hw_set==0) && (*argv[1]!='H'))
24                hw = get_hwtype("DFLT_HW");
25        printf("%s %s\n",ap,hw);
26        }
```

Figure 5.4: Simplified fragment of ARP in Coreutils/Net-tools–stable version.

```
Program P': New buggy inplementation
 1 #include<stdio.h>
 2 const char* get_hwtype (const char *name){
 3          return name;
 4 }
 5
 6 int main(int argc, char **argv){
 7         const char *hw = NULL;
 8         const char *ap = NULL;
 9         int hw_set = 0;
10         switch (*argv[1]){
11                 case 'A': case 'p':{
12                         ap = argv[2];
13                         if((hw_set==0) && (*argv[1]!='H'))
14                            hw = get_hwtype(argv[3]);
15
16                 break;
17                 }
18                 default: break;
19         }
20         printf("%s %s\n",ap,hw);
21         }
```

Figure 5.5: Simplified fragment of ARP in BusyBox–buggy version.

```
..main():::ENTER
this_invocation_nonce
0
argc
3
1
argv
4278190824
1
argv[..]
[ "./cuArp" ]
1

..get_hwtype():::ENTER
this_invocation_nonce
1
name
"DFLT_HW"
1
```

```
..main():::EXIT0
this_invocation_nonce
0
argc
3
1
argv
4278190824
1
argv[..]
[ "./cuArp" ]
1
return
13
1

..get_hwtype():::EXIT0
this_invocation_nonce
1
name
"DFLT_HW"
1
return
"DFLT_HW"
1
```

Figure 5.6: Declaration part of cuArp.dtrace file.

5.5.1 Trace File Generation

We run kvasir-dtrace ./ cuArp A inet and kvasir-dtrace ./ bbArp A inet in command line and produce cuArp.dtrace and bbArp.dtrace files respectively. The .dtrace file lists both variable declarations and values present from the given file. The declaration (.decls) parts of both files are shown in figure 5.6 and 5.7 respectively.

5.5.2 Invariant Detection

We run `java daikon.Daikon daikon-output / cuArp.dtrace` and `java daikon.Daikon daikon-output / bbArp.dtrace` in command line, that in turn produce cuArp.inv.gz and bbArp.inv.gz respectively, containing the list of invariants in compressed form. The figure 5.8 illustrates the invariants of the both files. The left portion of figure 5.8 presents the invariants found in cuArp.inv.gz whereas other portion presents that of bbArp.inv.gz.

```
..main():::ENTER
this_invocation_nonce
0
argc
3
1
argv
4278190808
1
argv[..]
[ "./bbArp" ]
1

..get_hwtype():::ENTER
this_invocation_nonce
1
name
nonsensical
2
```

```
..main():::EXIT0
this_invocation_nonce
0
argc
3
1
argv
4278190808
1
argv[..]
[ "./bbArp" ]
1
return
12
1

..get_hwtype():::EXIT0
this_invocation_nonce
1
name
nonsensical
2
return
nonsensical
2
```

Figure 5.7: Declaration part of bbArp.dtrace file.

```
================================|================================
..get_hwtype():::ENTER          |
name == "DFLT_HW"               |
================================|================================
..get_hwtype():::EXIT           |
return == "DFLT_HW"             |
return == orig(name)            |
================================|================================
..main():::ENTER                |..main():::ENTER
argc == 3                       |argc == 3
argv has only one value         |argv has only one value
argv[] == [./cuArp]             |argv[] == [./bbArp]
argv[] elements == "./cuArp"    |argv[] elements == "./bbArp"
size(argv[]) == 1               |size(argv[]) == 1
================================|================================
..main():::EXIT                 |..main():::EXIT
argv[] == [./cuArp]             |argv[] == [./bbArp]
argv[] elements == "./cuArp"    |argv[] elements == "./bbArp"
return == 13                    |return == 12
Exiting Daikon.                 |Exiting Daikon.
```

Figure 5.8: Invariants of *cuArp.inv.gz* (left) and *bbArp.inv.gz* (right).

```
<..get_hwtype():::ENTER>
  <name == "DFLT_HW" {1+}, null> (UInt,JM)
<..get_hwtype():::EXIT>
  <return == orig(name) {1+}, null> (Bin,JM)
  <return == "DFLT_HW" {1+}, null> (UInt,JM)
<..main():::ENTER>
  <null, size(argv[..]) == 1 {1+}> (UInt,MJ)
  <argv[..] elements == "./cuArp" {1+}, argv[..] elements == "./bbArp" {1+}>
  (UInt,DJJ)
  <argv[..] == [./cuArp] {1+}, argv[..] == [./bbArp] {1+}> (UInt,DJJ)
<..main():::EXIT>
  <return == 13 {1+}, return == 12 {1+}> (UInt,DJJ)
  <argv[..] elements == "./cuArp" {1+}, argv[..] elements == "./bbArp" {1+}>
  (UInt,DJJ)
  <argv[..] == [./cuArp] {1+}, argv[..] == [./bbArp] {1+}> (UInt,DJJ)
```

Figure 5.9: Invariant difference between *cuArp.inv.gz* and *bbArp.inv.gz*.

5.5.3 Invariant Comparison

Now we run java daikon.Daikon.diff.Diff cuArp.inv.gz bbArp.inv.gz in command line to produce the difference between the invariants of the two files. The difference of invariants is demonstrated in figure 5.9.

5.5.4 Output Analysis

If we look at figure 5.9 closely we can find out sharp difference of invariants between the two codes presented above. The <..function():::ENTER> and <..function():::EXIT> segments point out the difference between the invariants (function() is arbitrarily assumed for the two functions–*get_hwtype*() and *main*()) found in two *.inv* files. The inner portion of the ENTER and EXIT program shows the clear difference between the two programs. For

example if we consider <name="DFLT_HW" 1+, null>, it can be seen that the left portion of the comma (,) e.g., *name*=DFLT_HW presents the value of the invariant–*name* (in P). Whereas the value of name is null in P'. All other differences occurred in the invariant–difference file are basically due to systematic configuration. It means the value of variable *name* is DFLT_HW before entering into get_hwtype() in P but different in case of P'. The same happens after exiting get_hwtype(), returning DFLT_HW. This can be verified by the declaration part of *bbArp.dtrace* file (figure 5.7), where the ENTER and "EXIT" program points of *get_hwtype*() contains nonsensical values. This tells that the change of code at get_hwtype() in P leads to the change of value of variable name, hence the bug intrusion in P'.

Though, the manifestation of the bug in P' is at line number 20 (e.g., the output **inet NULL**) the root cause of the bug is present in get_hwtype() function, where the value of actual parameter passing to get_hwtype() is argv[3] contains NULL (as we put – ./bbArp A inet in command line). Hence it can be said that the bug is present at line number 14 in form of **argv[3]**. Figure 5.10 illustrates the whole concept.

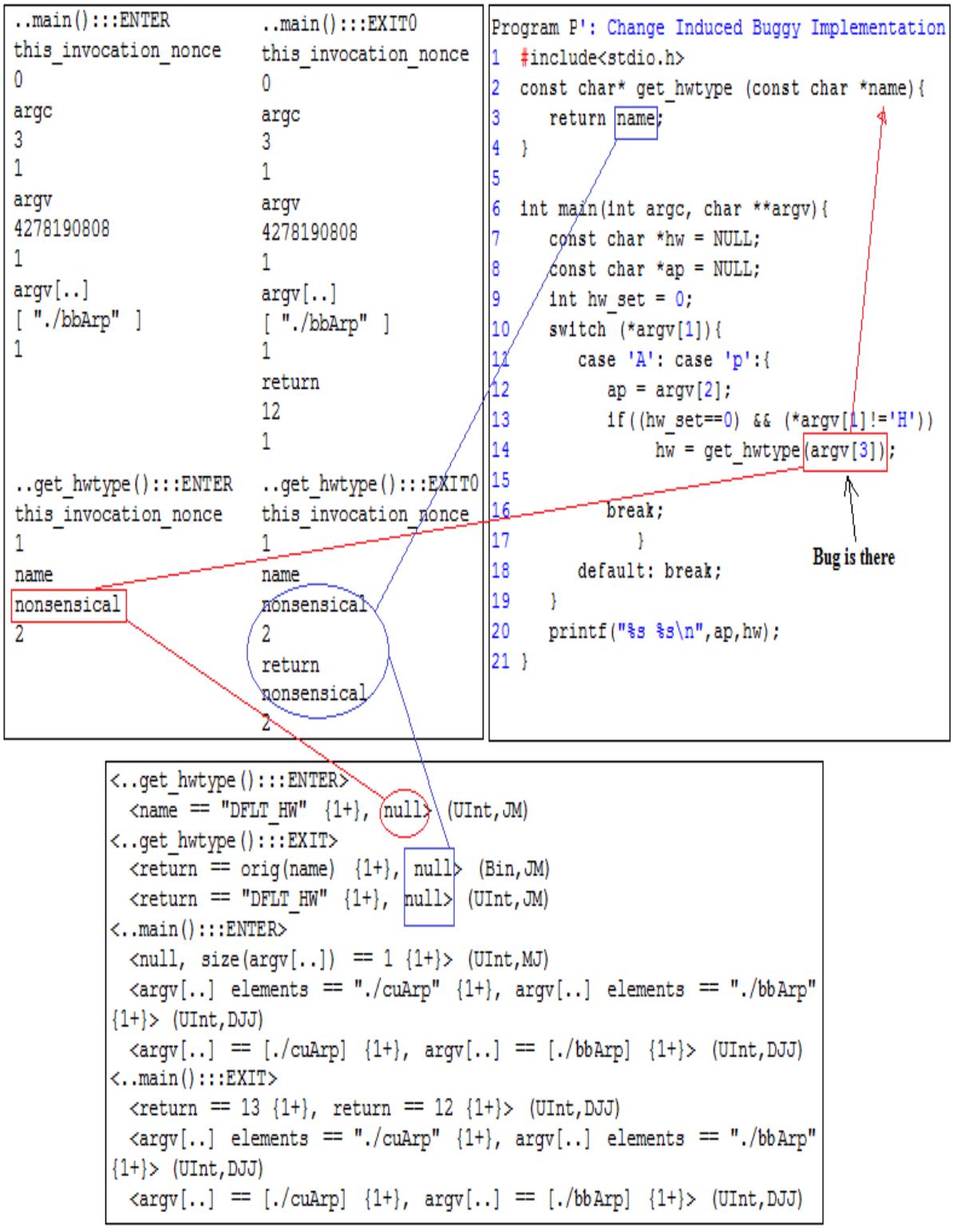

Figure 5.10: Illustration of source level bug localization incorporating invariant analysis.

Chapter 6

Object State Incorporated Debugging-OSiD

6.1 Introduction

Bug localization in object oriented program (oop) has always been an important issue in software engineering. In this chapter, we present a source level bug localization technique for object oriented embedded programs. Our proposed technique presents the idea of debugging an object oriented program in class level, incorporating the object state information into the class dependence graph (ClDG). Given a program (having buggy statement) and an input that fails and others pass, our approach uses concrete as well as symbolic execution to synthesize the passing inputs that marginally from the failing input in their control ow behavior. A comparison of the execution traces of the failing input and the passing input provides necessary clues to the root-cause of the failure. A state trace difference, regarding the respective nodes of the ClDG is obtained, which leads to detect the bug in the program.

6.2 ClDG

The class dependence graph (ClDG) represents the control and data dependencies within a class [3]. For a given class, the ClDG consists of a set of program dependence graphs (PDGs) [2] with additional edges to represent inter-procedural control and data dependencies. A statement in a procedure is represented by a statement vertex. Control and

data dependencies between program statements are represented by control dependence and data dependence edges, respectively. In this chapter we first take a buggy object oriented program and generate a state chart diagram and ClDG. After the diagrams are generated, we feed some test inputs into the test program, that results a fail and pass traces. Then one of the pass cases is selected that resembles most of the state information to that of failed one. Hence producing the bug report telling the position of bug inside the buggy program by involving state information.

6.3 Our Approach

We have named our technique object state-incorporated debugging for object oriented program (OSiD). Our technique is essentially based on first constructing models for test program (e.g., state chart and ClDG). Then a state transition table is created according to the state chart. In this approach we have incorporated state information to the ClDG. That means each node of ClDG is accompanied with a state (state of object of same class). Then a test suite $\{t_1, \ldots, t_i, \ldots, t_j, \ldots, t_k\}$ is applied as input into the test program. This results some passed and some failed test cases. Suppose the set of pass cases is $\{t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_k\}$ and fail set is $\{t_j\}$. Now at this point of OSiD, control dependence flow comparison is made between the elements of fail set $\{t_j\}$ and pass set $\{t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_k\}$. After the comparison, t_i is the found as sole test input that matches the most states to $\{t_j\}$ (in respect to control dependence flow). At this stage of OSiD State comparison occurs between $\{ t_i\}$ and $\{t_j\}$, resulting bug report, that points out the bug in source code level.

The important steps of OSiD approach have been shown in figure 6.1. The round corner blocks represent initial input test suite, control dependence flow comparison, state comparison, bug report etc..The rectangular blocks represent test object oriented program, ClDG, state chart, state transition table and other output test cases and sets etc. We now briefly discuss the different steps involved in our approach.

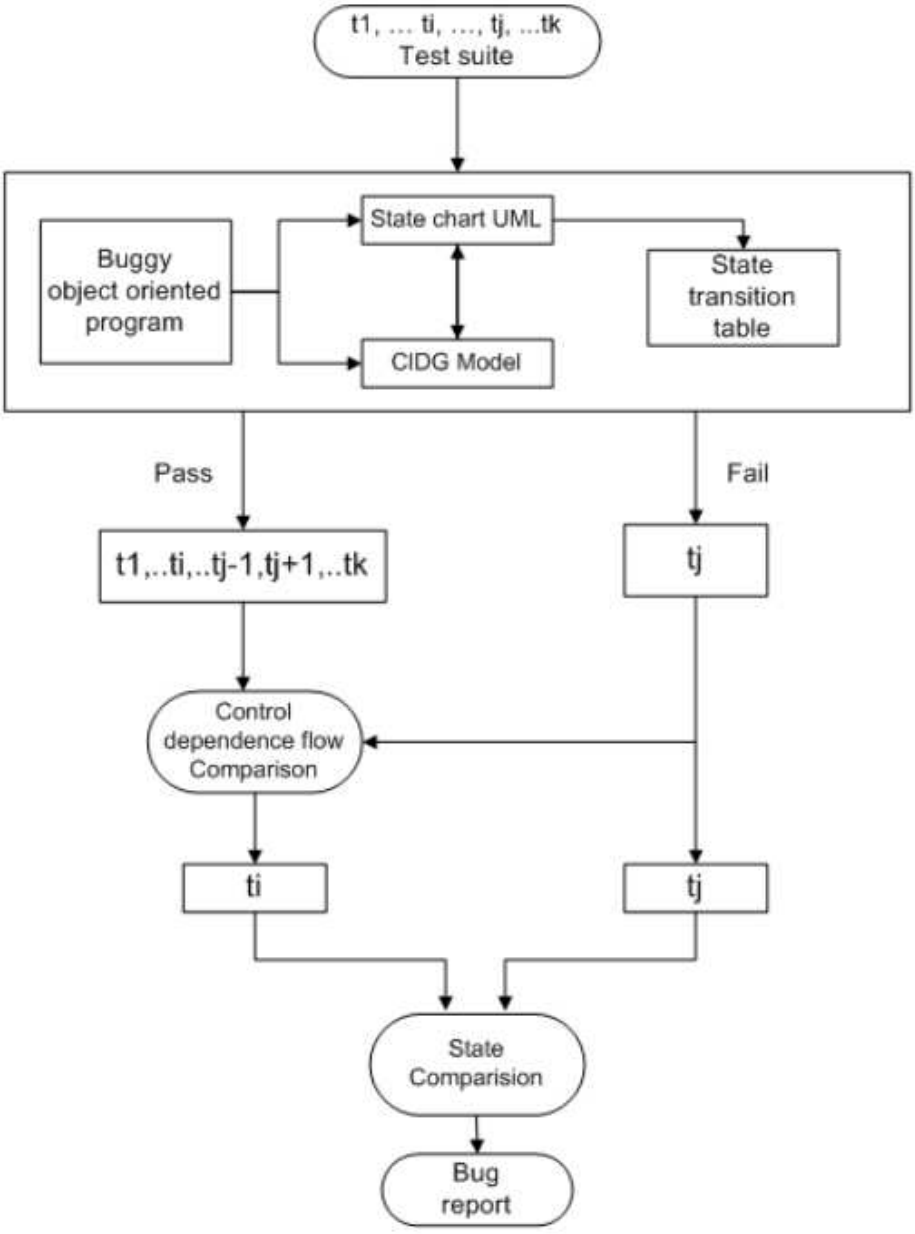

Figure 6.1: Overview approach of OSiD.

1. *The buggy object oriented program* is the source code written in C++ language. It is basically a test or buggy code where the logic of an elevator controller(embedded system program) is illustrated.

2. *The state chart unified modeling language* is the finite state machine model corresponding to the test program.

3. *The state transition table* is generated from state chart UML model. It contains various initial and final states with their condition and operation embedded in it.

4. The *ClDG* model represents the class dependence graph of the buggy program itself.

5. The Control dependence flow *comparison block* compares the control dependence flow between the failed and all other passed input.

6. The *State comparison* block compares the states of t_i and t_j, between the nodes of ClDG.

7. The *Bug report* is the final report regarding bug present inside the buggy program, which can point out the location of bug in source code level.

8. The $t_1, \ldots, t_i, \ldots, t_j, \ldots, t_k$ is the input test suite.

9. The $t_1, \ldots, t_i, \ldots, t_{j-1}, t_{j+1}, \ldots, t_k$ is the passed input of the previous said test-suite.

10. The t_j is the failed input from the input suite.

11. The t_i is the best chosen input from the set of passed inputs which belongs to initial test suite.

6.4 Bug Detection

Here we describe the experimentation on the elevator incorporating state information into the test program, written in C++.

6.4.1 Test OOP and Other Metrics

The figure 6.2 is the code snippet of an elevator controller written in C++. The controller has two main parts.

1. *Request Resolver*–resolves various floor requests into single requested floor.
2. *Control*–moves elevator to its requested floor.

1. *Description* :Elevator [1] moves either up or down to reach the requested floor. Once at the requested floor, open the door for at least 10 seconds, and keep it open until the requested floor changes. It is Ensured that the door is never open while moving. Elevator Does not change directions unless there are no higher requests when moving up or no lower requests when moving down. In this paper, we have taken the building to be three storied (having number of floors equal to three that is - ground -0, first-1, second-2) for simplicity.
2. *Prefixes introduced in program* :The numbers have been assigned sequentially to each statement in order they appear in the source code for identifying them in the ClDG. The prefixes S, E, CE and C denotes *statements, method entry, class entry* and *call nodes* respectively.
3. *Bug introduced in program* :In respect to our objective, we have introduced a bug inside the code. Line number S13 and S14 in *unitControl()* method of Control class. This results in door open when, any one requests a floor (which is 1 in this case) from a lower floor (say floor number 0). This repels the door to be open for 10000ms, thus not invoking the expected task (movement). This restricts the person (standing at ground floor) to go first floor. The code works well otherwise.
4. *Concurrency error ignorance* :This code snippet is purely an example of concurrency. Hence the hazards regarding concurrency such as deadlock, synchronization etc. are ignored at the time of documentation.

```
int up, down, open, floor=0, req;
CE1  class RequestResolver{

E2       int resolver(){
S3          while (1){
             ...
            //Get request from user
S4          cin>>req;
         }
     };
CE5  class Control{

E6       void unitControl(){

S7          int time= up = down = 0,open = 1;
S8          while (1) {
S9             while (req == floor); // Idle
S10             open = 0;
S11            if (req > floor) { up = 1; // Going Up
S12              while (req != floor) {
S13                if (req == 1)
S14                   goto P;
S15                floor++; } }

S16            else {down = 1;  // Going Down
S17              while (req != floor)
S18                floor--; }

S19         P: up = down = 0; open = 1;

               // Wait for 10000 ms  // Door Open
S20            for ( time=0; time < 10000; time++);}
               }
       };

E21  void main()
     {
S22     Control *c = new Control();
S23     RequestResolver *r = new RequestResolver();
S24     while(1) {   // Call concurrently:
C25     r -> resolver();
C26     c -> unitControl();}
     }
```

Figure 6.2: Buggy object oriented program snippet.

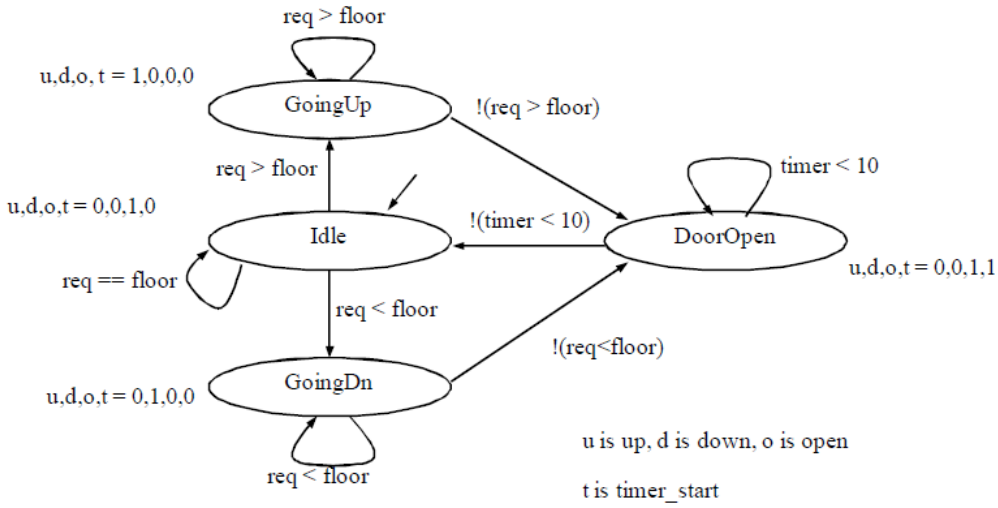

Figure 6.3: State chart of the test program.

6.4.2 State Chart

We have obtained a state chart diagram from figure 6.2 . The corresponding state diagram is shown in the figure 6.3. The chart shows four different states *Idle, Going Up, Going Down, Door Open* as the probable states. Possible transitions from one state to another is based on input (e.g., req > oor , req < oor etc.). Actions are occurred in each state (E.g., the GoingUp state; *u,d,o,t = 1,0,0,0* (up = 1, down, open, and timer start = 0).

6.4.3 State Transition Table

Figure 6.4 refers to the state transition table generated from Figure 6.3. The transition table contains *initial state, condition, operation/action, final state.* There is another state *Not Defined ND*, that can play an important role in providing object state information to those states which can not always satisfy the given state chart criteria. Going up =`u`,Going Down =`d`, Door open =`o`, timer start =`t` are different notions of actions, that frequently occur in each state of the class (object state information) under execution.

Initial State	Condition	Operation (Action)	Final State
Idle	req == floor	u,d,o,t =0,0,1,0	Idle
Idle	req > floor	u,d,o,t =1,0,0,0	Going Up
Idle	req < floor	u,d,o,t =0,1,0,0	Going Down
Going Up	req > floor	u,d,o,t =1,0,0,0	Going Up
Going Up	! req > floor	u,d,o,t =0,0,1,0	Door Open
Door Open	timer < 10	u,d,o,t =0,0,1,1	Door Open
Door Open	! timer < 10	u,d,o,t =0,0,1,0	Idle
Going Down	req < floor	u,d,o,t =0,1,0,0	Going Down
Going Down	! req < floor	u,d,o,t =0,0,1,0	Door Open

Figure 6.4: State transition table of produced state chart.

6.4.4 ClDG Representation of the Test Program

In this portion, we generate the ClDG corresponding to the test object oriented program (Figure 6.2). Along with this representation, we demonstrate another ClDG incorporated with state information of class *Control*, from the program. We choose this class regardless of other two modules of program (i.e class RequestResolver and void main()), as it plays the crucial role in elevator movement. Figure 6.5 and 6.6 presents the ClDG representation of the test program and its class *Control*, respectively.

6.4.5 Test Suite Deployment

In this portion, we feed the test program with input test suite. This results in a test suite decision table (TSD). This table presents the combination of all floors and corresponding requests from a three storied building. This table shows all the nodes which belong to the class *Control*. The + sign represents the full execution of the respective node, while − sign tells about the bypassing nodes. Finally the input combination outputs the fail or pass mark, shown in figure 6.7.

All test cases involving <floor, req> are shown in the figure 6.7. From this, we got the clear view of control dependence flow in respect to the nodes of ClDG. The <0, 1>

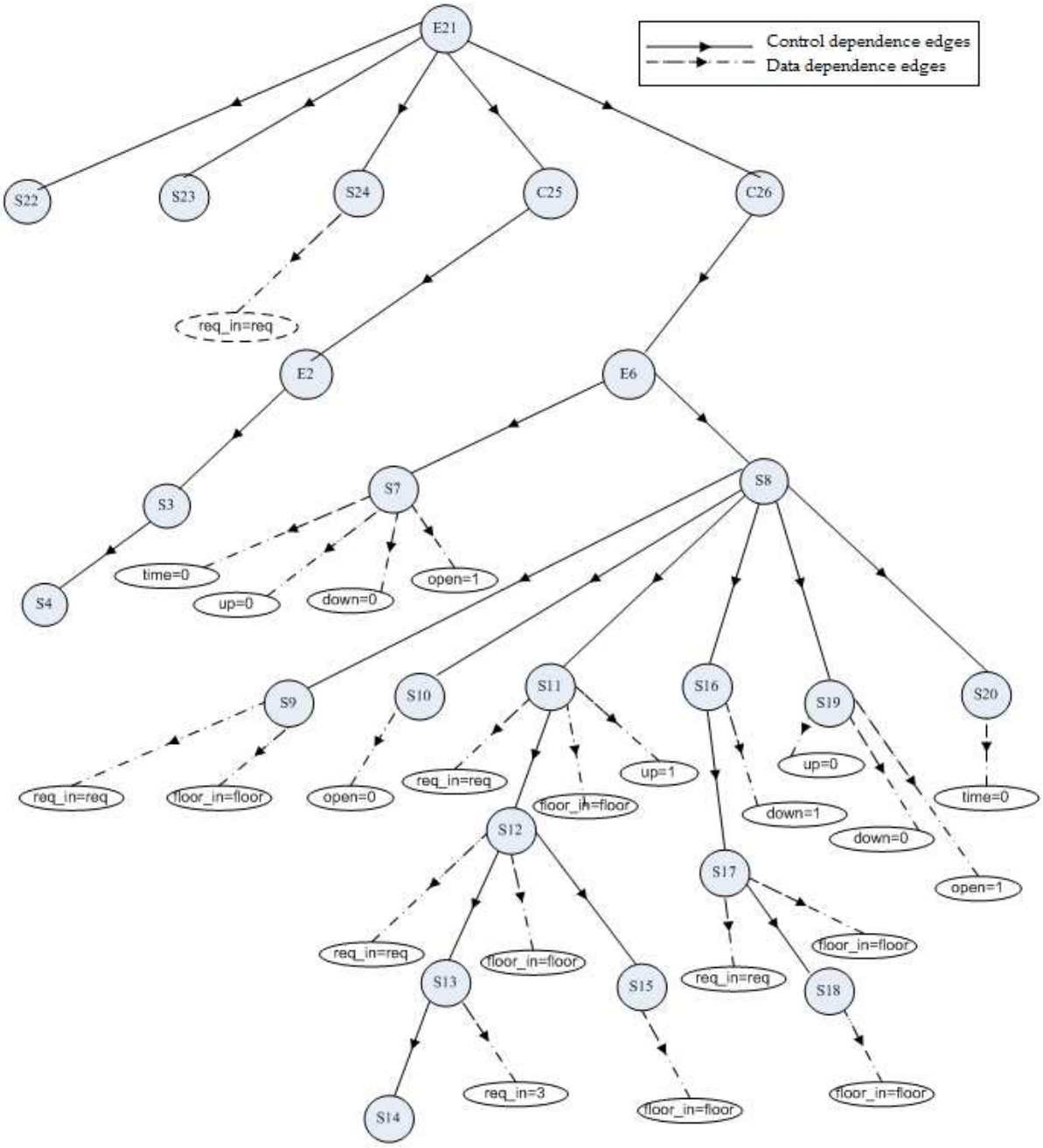

Figure 6.5: ClDG of the whole program.

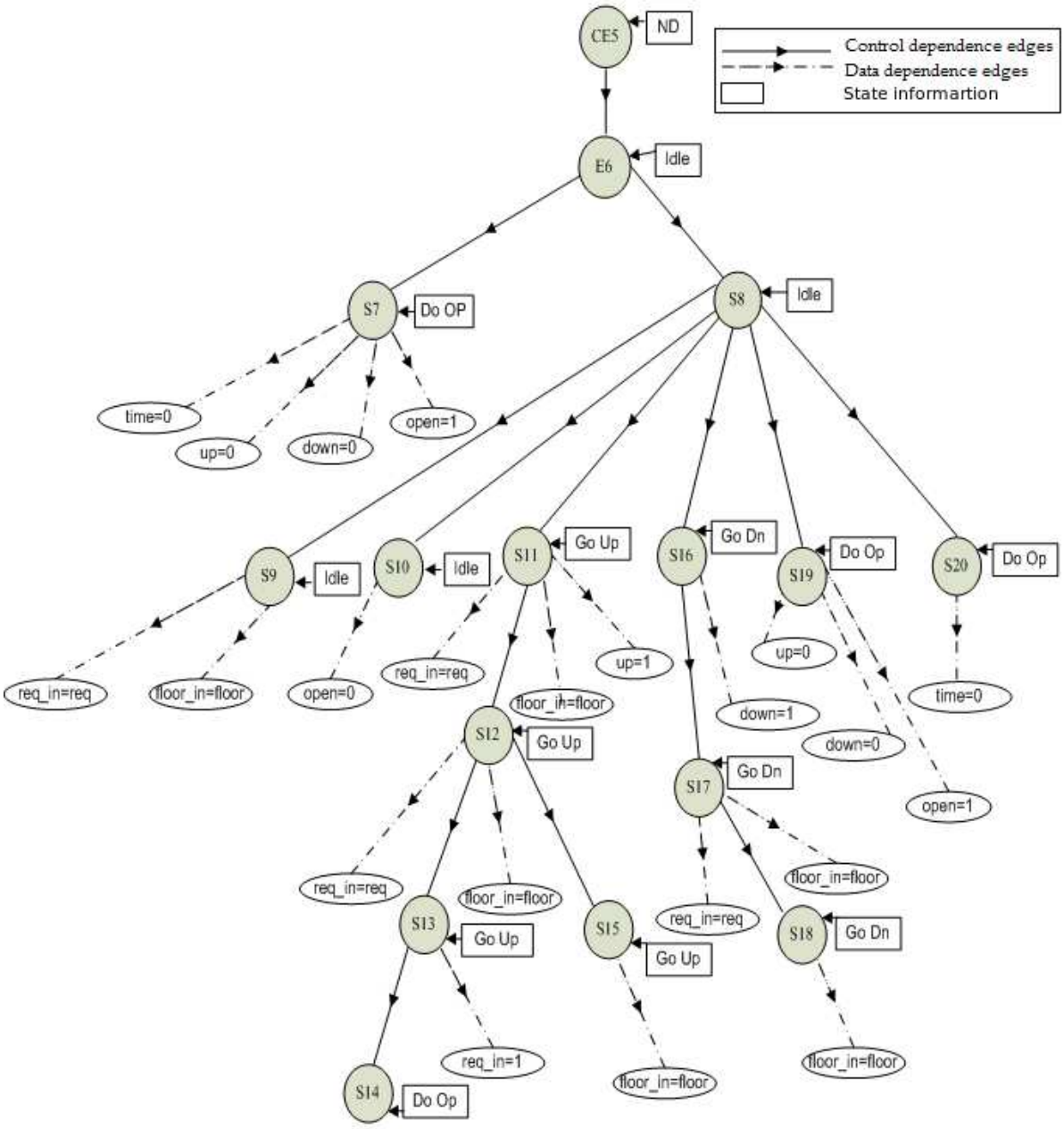

Figure 6.6: State information incorporated ClDG of class *Control*.

floor	req	CE5	E6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20	Test
0	0	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	Pass
0	1	+	+	+	+	-	+	+	+	+	+	-	-	-	-	+	+	Fail
0	2	+	+	+	+	-	+	+	+	-	-	+	-	-	-	+	+	Pass
1	0	+	+	+	+	-	+	-	-	-	-	-	+	+	+	+	+	Pass
1	1	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	Pass
1	2	+	+	+	+	-	+	+	+	-	-	+	-	-	-	+	+	Pass
2	0	+	+	+	+	-	+	-	-	-	-	-	+	+	+	+	+	Pass
2	1	+	+	+	+	-	+	-	-	-	-	-	+	+	+	+	+	Pass
2	2	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	Pass

+ Statement executed truly - Statement did not executed

Figure 6.7: Test suite decision (TSD) table for class *Control*.

floor	req	CE5	E6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20	Test
0	1	Nd	Id	Do	Id	-	Id	Gu	Gu	Gu	Do	-	-	-	-	Do	Do	Fail
1	2	Nd	Id	Do	Id	-	Id	Gu	Gu	-	-	-	-	-	-	Do	Do	Pass

State information at each node: Id – Idle Gu – Going up Gd – Going down Do – Door open
Nd – Not defined

Figure 6.8: State comparison between the failure and passing test cases.

input results in failure, whereas others pass. Analyzing the whole test suit decision table, it was found that input <0, 1> and <1, 2> are closely matched to each other (in respect of control dependence flow).

6.4.6 State Information Comparison

Here we present the state information comparison between the pair of test inputs shown in figure 6.8. The failed input <0, 1> and the passing input <1, 2>. The figure 6.8 illustrates the simple process. *Nd, Id, Do, Gu, Do* represents *Not defined, Idle, Door open, Going up, Going down* respectively. From the figure 6.8, it is clear to understand that the two inputs <0, 1> and <1, 2> differs at node number **S13** and **S14**, in respect of states (Gu at S13 and Do at S14), shown in *rounded* form.

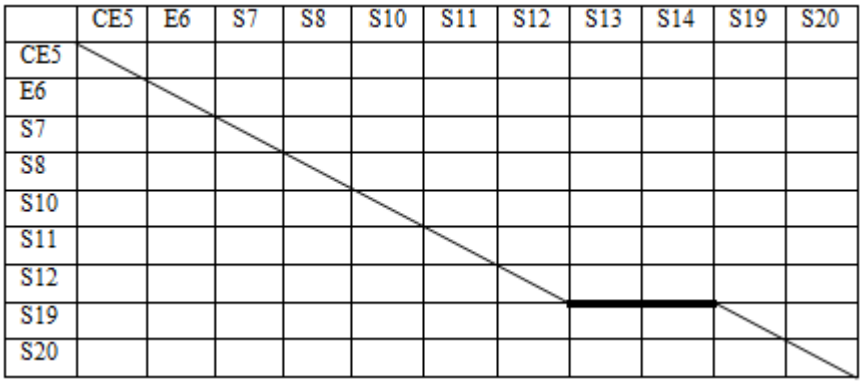

Figure 6.9: State comparison matrix (SCM) illustrating the bug.

6.4.7 State Comparison Matrix

State comparison matrix (SCM) is a matrix that represents the state alignment of failing input to the passing one. The figure 6.9 shows the state comparison between <0, 1> and <1, 2>. The left most column presents the state information of passing input (<0, 1>). Whereas the upper most row presents the state information of failing input (<1, 2>). The matrix basically shows a straight line starting from upper left most corner to the lower right most. The line represents the state alignment between these two test inputs. The line is at an slope through out its way, leaving the S13 and S14 numbered columns.

6.4.8 Source Level Bug Localization

Now if, we take the information from 6.4.7 and search the reason of the misbehavior of the line in the matrix (flat line), we found that a change of states occurred during the execution to the respective line numbers of the given code. While investigating this reason, we found that line number **13** of the program involves an *if* statement, which when executes the bug got infiltrated inside the program, making it **buggy**. The *if* block satisfies when the *req==1* condition satisfies, resulting the control flow to jump at line number *19*. This keeps the door open, making the Door open state true. The Figure 6.10 shows the buggy segment in side the code.

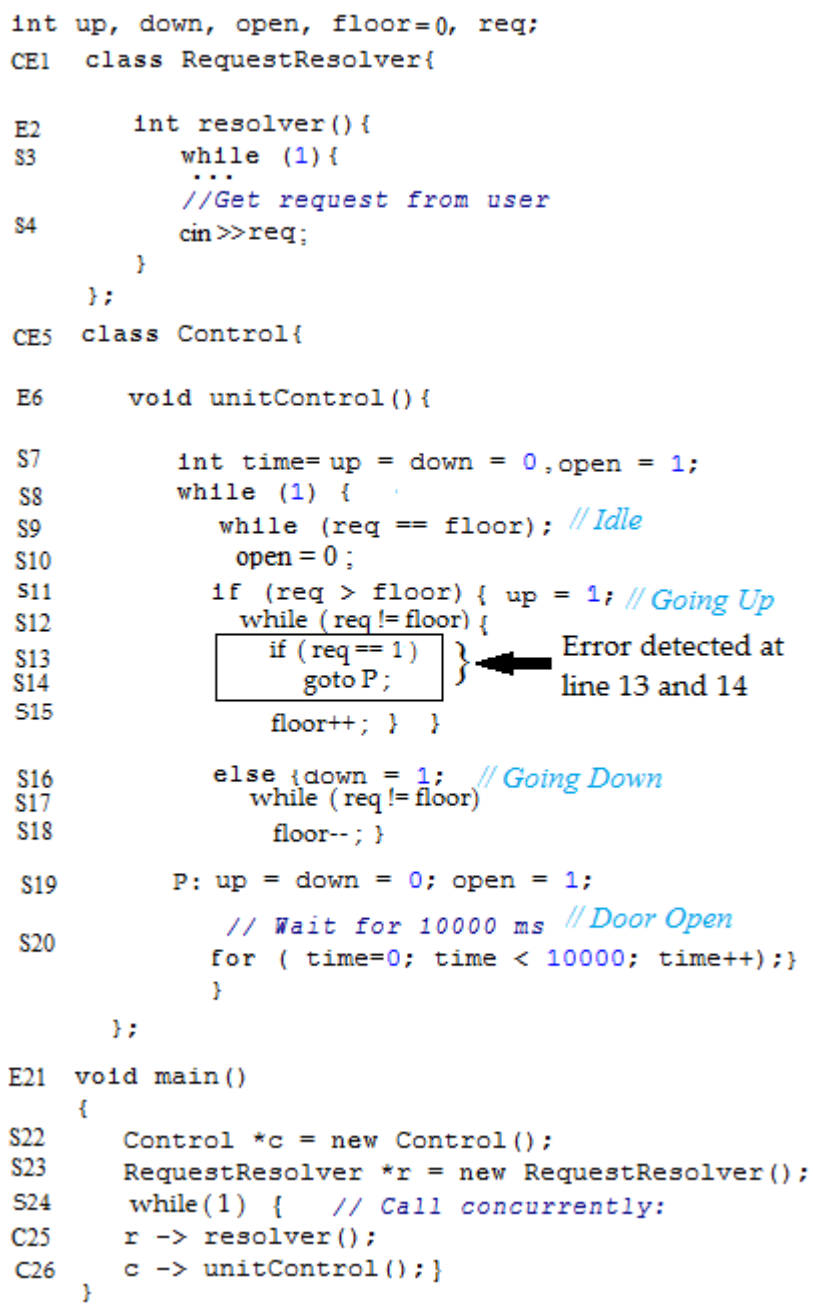

Figure 6.10: Bug localization in source code level.

Chapter 7

Conclusion and Future Work

In this thesis, we have presented a few bug tracking techniques and methodologies, specially memory related. We have tested our techniques on BusyBox, which is a very well known embedded Linux distribution, and an object oriented code snippet, written in C++. We have localized root cause of two memory related errors by Valgrind. We have performed invariant analysis on a block of code from BusyBox by Daikon. As a last work, have presented an approach to debug any object oriented embedded program incorporating object state information into class dependence graph.

As a whole, we have presented three types of error detection possibilities that can be applied on embedded softwares to seek for the bugs.

In future, we want to develop a rigorous technique that will automatically detect the above said errors providing few more critical parameters.

We would like to put down an automated code which will ease the job of debugging.

List of Publications

1. Partha Pratim Ray, Ansuman Banerjee, Debugging Memory Issues In Embedded Linux: A Case Study, In IEEE Techsym 2011.

2. Partha Pratim Ray, Dr.Ansuman Banerjee, Banibrata Bag, Debugging Invariant Issues In Pseudo Embedded Program: an Analytical Approach, In International Journal of Computer Science and Information Technologies, ISSN: 09759646, Vol. 2 (2) , 2011, pp. 780-785.

References

[1] Embedded Systems Design: A Unied Hardware/Software Introduction, (c) 2000 Vahid/Givargis.

[2] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9(3):319349, Jul 1987.

[3] Gregg Rothermel and Mary Jean Harrold. Selecting regression tests for object- oriented software. In Proceedings of the International Conference on Software Maintenance - 1994, pages 1425, Victoria, BC, Canada, Sep 1994.

[4] J. H. Perkins, M. D. Ernst, Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants, In SIGSOFT'04/FSE12.

[5] David Schuler, Valentin Dallmeier, and Andreas Zeller, Efcient Mutation Testing by Checking Invariant Violations, In ISSTA 2009, July 19-23, USA.

[6] N. Nethercote,J. Seward, How to Shadow Every Byte of Memory Used by a Program, In VEE 2007, June 13-15, san Diego, USA.

[7] A. X. Zheng, B. Liblit, M. I. Jordan, A. Aiken, Statistical Debugging of Sampled Programs.

[8] Daikon, http://pag.csail.mit.edu/daikon/.

[9] Farinaz Koushanfar, Darko Kirovski, Inki Hong, Miodrag Potkonjak, Marios C. Papaefthymiou, Symbolic Debugging of Embedded Hardware and Software, In IEEE Transactions On Computer-Aided Design Of Integrated Circuits and Systems, VOL. 20, NO. 3, MARCH 2001.

[10] Neelam Gupta, Haifeng He , Xiangyu Zhang, Rajiv Gupta, Locating Faulty Code Using Failure-Inducing Chops, In ASE 2005, November 7-11, USA.

[11] R. F. Abreu, Spectrum-based Fault Localization in Embedded Software, ISBN 978-90-79982-04-2, 2009.

[12] S. Hamgal,S. Narayanan, N. Chanda, S. Chakravorty, IODINE: A Tool to Automatically Infer Dynamic Invariants for Hardware Designs, In DAC 2005, June 13-17, USA.

[13] Tao Wang, Abhik RoyChoudhury, Automated Path Generation for Software Fault Localization., In In Proceedings of ASE 2005, November 7-11, USA.

[14] Liang Guo, Abhik Roychoudhury, and Tao Wang, Accurately Choosing Execution Runs for Software Fault Localization.

[15] Valgrind, http://valgrind.org/.

[16] Ansuman Banerjee, Abhik Roychoudhury, Johannes A. Harlie, Zhenkai Liang , Golden Implementation Driven Software Debugging, In In Proceedings of FSE-18, 2010.

[17] Cristian Cadar, Daniel Dunbar, Dawson Engler , KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, In proceedings of 8th USENIX Symposium on Operating Systems design and Implementatoin.

[18] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang , Test Generation to Expose Changes in Evolving Programs, In Proceedings of ASE-10, 2010.

[19] Maximilian Stoerzer, Barbara G. Ryder and Frank Tip, Xiaoxia Ren, Finding Failure-Inducing Changes in Java Programs using Change Classication, In SIGSOFT'06/FSE-14, USA.

[20] Andreas Zeller, Isolating Cause-Effect Chains from Computer Programs, In SIGSOFT 2002/FSE-10 , USA.

[21] Andreas Zeller, Holger Cleve, Delta Debugging: Automating the Scientic Method.

[22] BusyBox, http://busybox.net/.

[23] Philip J. Koopman, Jr.,Embedded System Design Issues (the Rest of the Story),In proceedings of the International Conference on Computer Design (ICCD 96).

[24] Embedded System, http://en.wikipedia.org/wiki/Software_system.

[25] Michael Barr. "Embedded Systems Glossary". Netrino Technical Library. http://www.netrino.com/Embedded-Systems/Glossary. Retrieved 2007-04-21.

[26] Heath, Steve (2003). Embedded systems design. EDN series for design engineers (2 ed.). Newnes. p. 2. ISBN 9780750655460, "An embedded system is a microprocessor based system that is built to control a function or a range of functions."

[27] Michael Barr; Anthony J. Massa (2006). Introduction to Programming embedded systems: with C and GNU development tools. O'Reilly. pp. 12. ISBN 9780596009830.

[28] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, Communications of the ACM, Vol. 21, No. 7, July, 1978.

[29] G. Kahn, The Semantics of a Simple Language for Parallel Programming, Proc. of the IFIP Congress 74, North-Holland Publishing Co., 1974.

[30] A. Benveniste and P. Le Guernic, Hybrid Dynamical Systems Theory and the SIGNAL Language, IEEE Trans. on Automatic Control, vol. 35, no. 5, pp. 525- 546, May 1990.

[31] R. Ben-Natan, CORBA: A Guide to Common Object Request Broker Architecture, McGraw-Hill, Inc., ISBN 0-07-005427-4, 1995.

[32] B. Selic, G. Gullekson, and P. Ward, Real-Time Object-Oriented Modeling, John Wiley & Sons, New York, NY 1994.

[33] D. Harel and A. Pnueli, On the Development of Reactive Systems, in Logic and Models for Verification and Specification of Concurrent Systems, Springer Verlag, 1985.

[34] G. Berry, Real Time programming: Special purpose or general purpose languages, in Information Processing, Ed. G. Ritter, Elsevier Science Publishers B.V. (North Holland), vol. 89, pp. 11-17, 1989.

[35] A. Benveniste and G. Berry, The Synchronous Approach to Reactive and Real-Time Systems, Proceedings of the IEEE, vol. 79, no. 9, 1991, pp. 1270-1282.

[36] G. Berry and G. Gonthier, The Esterel synchronous programming language: Design, semantics, implementation, Science of Computer Programming, vol. 19, no. 2, pp. 87-152, 1992.

[37] Edward A. Lee, Embedded Software, Advances in Computers (M. Zelkowitz, editor) 56, Academic Press, London, 2002.

[38] Smith, Mark, What gets measured gets done, Cobalt Group, http://mspresso.wordpress.com/2008/03/31/what-gets-measured-gets-done/.

[39] TypeI and TypeII errors, http://en.wikipedia.org/wiki/TypeI_and_TypeII_errors.

[40] Software bug, http://en.wikipedia.org/wiki/Software_bug.

[41] Debugging, http://en.wikipedia.org/wiki/Debugging.

[42] Grace Hopper, http://foldoc.org/.

[43] Shields, Tyler, Anti-Debugging Series - Part I.

[44] The Chinook Helicopter Disaster, http://www.ccsr.cse.dmu.ac.uk/resources/general/ethicol/Ecv1

[45] Software bugs cost US economy dear, http://www.nist.gov/public_affairs/releases/n02-10.htm.

[46] Excluding other sorts of intentional misrepresentation such as fraud, http://en.wikipedia.org/wiki/Type_I_and_type_II_errors-cite_ref-4.

[47] The magnitude of the error is contingent upon the amount by which the observation differs from its expected value, http://en.wikipedia.org/wiki/Type_I_and_type_II_errors-cite_ref-multiple_5-0.