ISSN:0975-9646

Debugging Invariant Issues in Pseudo Embedded Program: an Analytical Approach

Partha Pratim Ray[1], Dr. Ansuman Banerjee[2], Banibrata Bag[3]
[1,3]*Department of Electronics and Communication Engineering*
Haldia Institute of Technology
Haldia, Purba Medinipur-721657
West Bengal, India

[2]*Advanced Computing and Microelectronics Unit*
Indian Statistical Institute
Kolkata-700108, West Bengal, India

Abstract—**Debugging is an unavoidable and most crucial aspect of software development life cycle. Especially when it comes the turn of embedded one. Due to the requirements of low code size and less resource consumption, the embedded softwares need to be upgraded all the time involving obvious change of code during development phase. This leads the huge risk of intrusion of bugs into the code at production time. In this paper we propose an approach of debugging embedded program in pseudo format, incorporating invariant analysis. Our methodology works on top of Daikon, a popular invariant analyzer. We have experimented with a simplified code snippet [1], used during debugging a reported error in BusyBox which is a de-facto standard for Linux in embedded systems.**

Keywords- *Debugging, embedded system, embedded linux, busybox, invariant analysis*

I. INTRODUCTION

Today's technological growth in the areas of Very Large Scale Integration (VLSI), and system programming brought down the so called problematic approach to design the sophisticated program on the hardware base, allowing the programmers to develop large hard core pieces of program in minimal time. Embedded programs are mandated to some special attentions (e.g. low code size, lower memory foot-print etc.) as compared to usual general purpose programs (softwares). All these things have made embedded programs to get designed and developed without sufficient sanity checking norms (exceptions, signal handlers, assertions etc.). This results in huge probability in bug intrusion into the embedded code the production time.

A software bug [2] is an error, flaw, mistake, undocumented feature that prevents it from behaving as intended (e.g. producing an incorrect result). Bugs normally arise due to the mistakes and wanted/unwanted logical errors produced by human being (programmer). It should be noted that reproduction of a bug may be very different in various angles than to the actual one presented in the code. So the Debug method must be very efficient to trace back the bug from its physical manifestation. A bug report should be able to seek out the root cause of the manifested bug.

We employ our method on a code snippet (simplified form) [1], developed from the reported error (arp utility) [6] in Busy Box [BusyBox provides many of the standard utilities but has a smaller code size], the de-facto standard for Linux in embedded devices. We imply the test input on this code and feed the output into Daikon to produce a set of invariants. We take another modified (buggy) version with some modifications of the same code and do the same. At this point we compare and analyze the two sets of invariants and detect the source level root cause of the bug in the buggy version.

This paper is organized as follows: Section II presents related work. Section III presents an overview of our approach. Section IV presents facts about Daikon and invariant. Section V presents the idea of BusyBox behind our approach. Section VI describes our experimentation done with Daikon. Section VII presents the analysis. Section VIII concludes this paper.

II. RELATED WORK

One of the first efforts for debugging program changes is [4]. This paper identifies the changes across program versions and searches among subsets of this change set to identify which changes could be responsible for the given observable error. In evolving program debugging, a buggy program version is simultaneously analyzed with older stable program version. A very recent paper [5] has proposed an approach to debug the various memory related issues in embedded Linux. This methodology implements the previously reported bugs [6] in BusyBox, a de-facto standard for Linux in embedded devices. This methodology experimented with Valgrind [7] and Daikon [3]. Though, the paper pointed out the root cause of bugs in source code level, but failed to produce any promising result using Daikon.

IODINE [15] is such a tool which can automatically extract likely design properties such as state machine protocols, request-acknowledge pairs, and mutual exclusion between signals from design simulations. This literature showed that the dynamic invariant detection for hardware designs can infer relevant and accurate properties. In 2009, a paper proposed the

DARWIN [8] approach for debugging program versions. DARWIN performs dynamic symbolic execution along the execution of a given test input in two programs. DARWIN method is basically suited for debugging branch errors (or code missing errors where the missing code contains branches). Dynamic slicing [9] has so far been studied as an aid for program debugging and understanding. A recent work [10] uses dynamic program dependencies to seek the involved parts of an input that are responsible for a given failed output. Research such as [11,12] combine symbolic execution and dependency analysis for test suite augmentation.

Another recent literature [1] proposed the possibility of using the golden implementation as a reference model in software debugging. This paper experimented involving the BusyBox embedded Linux utilities while treating the GNU Core Utilities as the golden or reference implementation. This debugging method consisted of dynamic slicing with respect to the observable error in both the implementations (the golden implementation as well as the buggy software). During dynamic slicing this methodology also performed a step by step weakest precondition computation of the observable error with respect to the statements in the dynamic slice. The formulae computed as weakest pre-condition in the two implementations are then compared to accurately locate the root cause of a given observable error.

Another approach used Daikon [3] in its learning component to analyze the various invariants present in the code. HeapGuard and determina Memory Firewall [13] helped in monitoring phase incorporating the monitor to detect a failure and the failure location.

III. Overview of Our Approach

In this section we present our approach to debug the embedded program. To proceed further we need to clarify the objective of this movement.

A. Objective

Embedded programs are developed to perform a predefined specific task. In this respect it should be pointed that with the rapid development of embedded programming structure and huge market requirement, 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 (buggy). Fig. 1 shows the same in pictorial form.

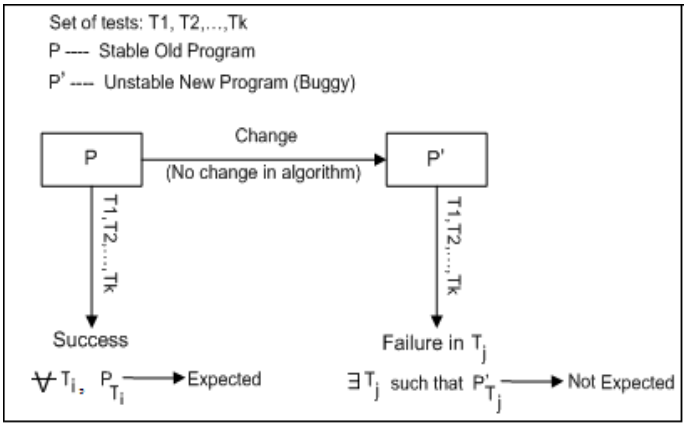

Fig. 1 Test input feeding into stable and buggy program.

The figure above 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 fig. 2 illustrates the same.

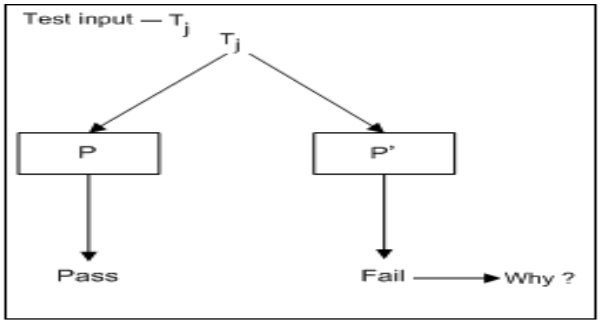

Fig. 2 Test input T_j passes in P but fails in P'.

B. Overview

We go through a dedicated methodology that seeks out the root cause of change induced bugs (bugs that introduced due to change of code). Our methodology works on top of Daikon, a popular invariant analyzer. Fig. 3 presents the whole of our thought to debug the change induced bugs in embedded program.

In fig. 3 we have shown our detail methodology to debug an embedded program incorporating 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'. In the mentioned *successful run* and *failed run*, we want to mean 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.

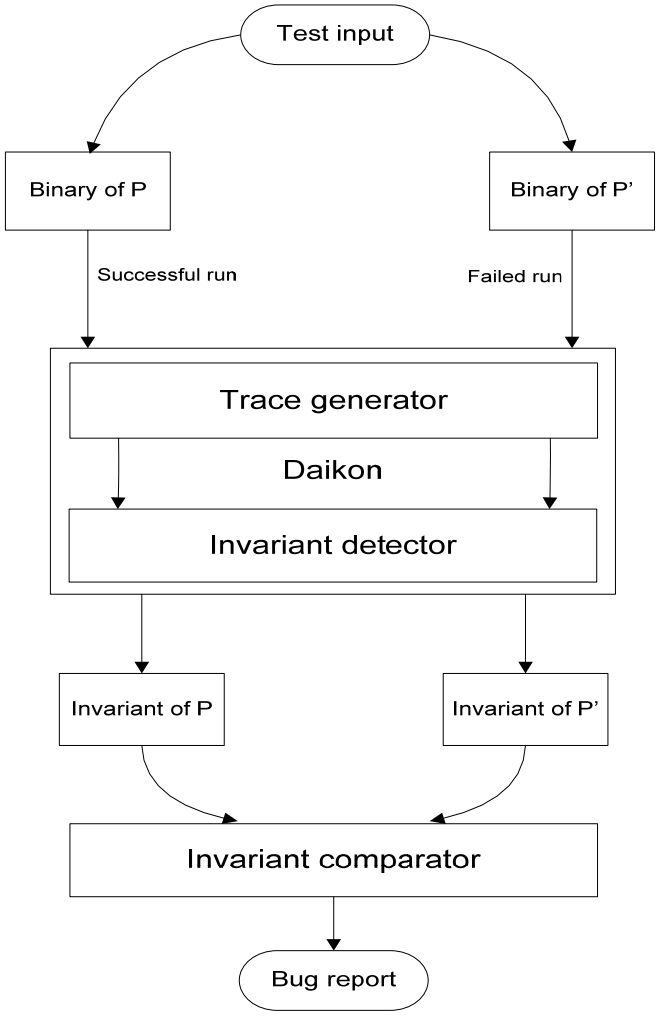

Fig. 3 Overview of invariant analysis incorporated embedded program debugging approach.

The next phase of the methodology relies heavily on Daikon. Where we feed the two binaries in the front-end (insrtumenter) (*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.

Hence after, we proceed with the bug report to analyze the invariant/s that was/were not present in any of the two programs or have has/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 invariant/s in the code.

IV. FACTS ABOUT DAIKON AND INVARIANT

This section tells about Daikon and the very aesthetic aspect of our approach --- invariant.

A. Daikon

Daikon is an implementation of dynamic detection of likely invariants; that is, 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. That data can come from any source, but Daikon is typically used to find invariants over variable values in running programs. A front end is a tool that converts data from some other format into Daikon's input format. The most common type of front end is an instrumenter, which causes your program to output a '*.dtrace*' file that Daikon can process. Daikon runs on Java Virtual Machine (JVM).

Daikon comes with two front ends for the C language: Kvasir and Mangel-Wurzel. Kvasir only works under the Linux operating system, and it works only on "x86" and "x86/64" processors. Mangel-Wurzel lacks some tracing features related to arrays and nested structs, and requires Purify. Another front end is there for Java language --Chicory. It executes Java programs, creates data trace ('.dtrace') files, and optionally runs Daikon on them. We choose Kvasir in this paper as it is freely available and comes with distribution of Daikon package.

Daikon invariant detector --- '*java daikon.Daikon'* takes different '*.dtrace'*, '*.decl*' files to produce '*.inv*' (invariant) file. There are various control and debugging options available with Daikon that can be used for sake of better results in different angles. Along with these advantages, Daikon has its own way of testing mechanisms as unit testing and regression testing.

B. Invariant

An invariant is a property that holds at a certain point or points in a program (An invariant is simply an expression that evaluates to "true" on all executions (paths) of the 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.

C. 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 [14].

D. *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 postconditions. 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 postconditions is formally *nonsensical.* Examples include: x when x is uninitialized or deallocated, x.y when x is null (uninitialized or deallocated), a[i] when i is outside the bounds of a (uninitialized or deallocated, or a is null).

V. BUSYBOX CONCEPT BEHIND OUR APPAROACH

[1] implemented golden implementation driven debugging method on BusyBox —the de-facto standard for Embedded Linux devices. It provides many of the standard Linux utilities, but has a smaller code size (size of the executable) than the GNU Core Utilities, net-tools and procps. This paper employed debugging methods on BusyBox version 1.4.2 and 1.16.0, to find the root causes of errors that have previously been reported in literature [6]. We are interested in '*arp –Ainet* 'bug to demonstrate our approach. In standard Linux distribution *arp* is found in network utility which manages processor's network neighbor cache. It can add or delete the 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*. '*arp –Ainet'* in command line argument executes well when run in GNU environment, but produces wrong output when run in BusyBox—1.4.2. At this point of view we can assume the source program (arp.c) of the above said bug-- *arp* in GNU environment to be old 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 [1] to implement our methodology in simpler way. We have termed it P, whereas the code snippet that we have developed from P (changing a few line of code) is P'. Both the programs P and P' act as the stable and buggy respectively. Fig. 4 and fig. 5 represents P and P' respectively.

VI. EXPERIMENT ON DAIKON

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 (debug build) 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.

Program P: Old Stable Implementation

```
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 }
```

Fig. 4 Simplified fragment of ARP in Coreutils/Net-tools – stable version.

Program P': Change Induced Buggy Implementation

```
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 }
```

Fig. 5 Simplified fragment of ARP in BusyBox– buggy version.

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 [1].

A. *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 in the given file. The declaration (*.decls*) parts of both files are shown in fig. 6 and 7 respectively.

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

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

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

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

Fig. 6 Declaration part of *cuArp.dtrace* file.

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

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

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

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

Fig. 7 Declaration part of *bbArp.dtrace* file.

B. *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* repectively, containing the list of invariants in compressed format. Fig. 8 illustrates the invariants of the both files. The left portion of fig. 8 presents the invariants found in *cuArp.inv.gz* whereas other portion presents that of *bbArp.inv.gz.*

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

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

Fig. 8 Invariants of *cuArp.inv.gz* (left) and *bbArp.inv.gz* (right).

C. *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 fig. 9.

```
<..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)
```

Fig. 9 Invariant difference between *cuArp.inv.gz* and *bbArp.inv.gz.*

VII. ANALYSIS

If we look at fig. 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_hwtype0* 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 (fig. 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 output or 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 commandline). Hence it can be said that the bug is present at line number 14 in form of *argv[3]*. Fig. 10 illustrates the whole concept.

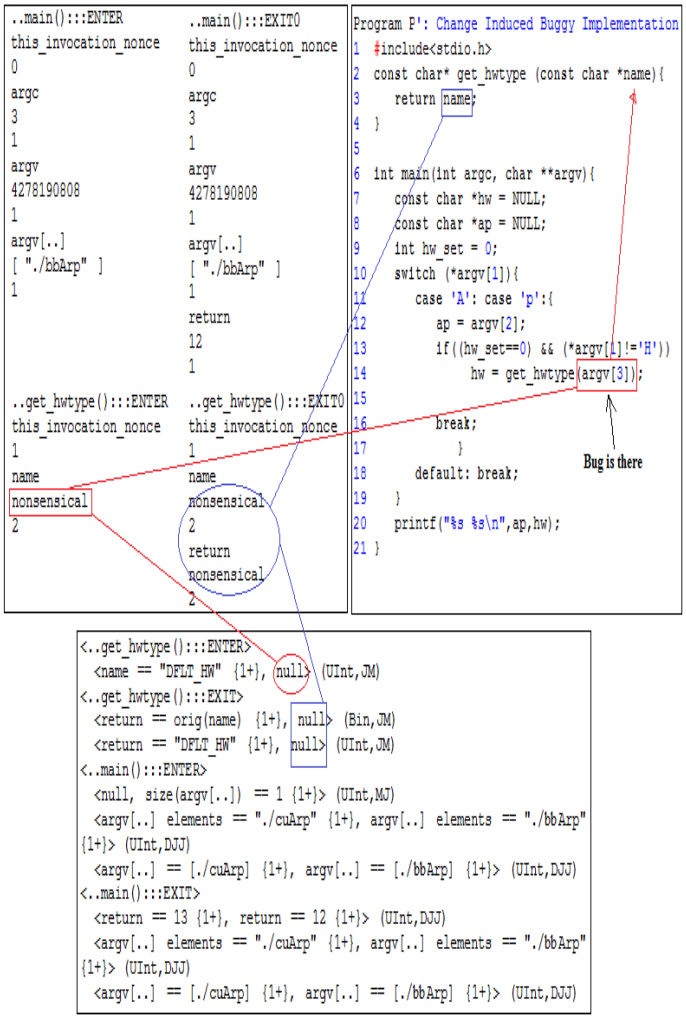

Fig. 10 Illustration of source level bug localization incorporating invariant analysis.

VIII. CONCLUSION

In this paper we propose a methodology for debugging errors in embedded program. Our methodology works on top of Daikon, an invariant analyzer. We have experimented with a published error in BusyBox in pseudo format and found promising results. Currently, we are investigating to develop software which will automatically detect the errors present in embedded program correlating the invariant differences after generating the invariants produced by Daikon.

REFERENCES

[1] A. Banerjee, Abhik R. Choudhury, Johannes A. Harlie, Zhenkai Liang, Golden implementation driven software debugging, in FSE-18, 2010.
[2] Software bug. http://en.wikipedia.org/wiki/Software bug.
[3] Daikon. http://pag.csail.mit.edu/daikon/.
[4] A. Zeller. Yesterday my program worked, today it does not. Why? In ESEC-FSE, 1999.
[5] P. P. Ray, A. Banerjee, Debugging memory issues in embedded linux: a case study, In IEEE Tech sym, 2011.
[6] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI, 2009.
[7] Valgrind. http://valgrind.org/.
[8] Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. DARWIN: An approach for debugging evolving programs. In ESEC-FSE, 2009.
[9] B. Korel and J. W. Laski. Dynamic program slicing. Information Processing Letters, 29(3):155163, 1988.
[10] J. Clause and A. Orso. Penumbra: Automatically identifying failurerelevant inputs using dynamic tainting. In ISSTA, 2009.
[11] D. Qi, A. Roychoudhury, and Z. Liang. Test generation to expose changes in evolving programs. In ASE, 2010.
[12] R. Santelices, P. Chittimalli, T. Apiwattanapong, A. Orso, and M. Harrold. Test-suite augmentation for evolving software. In ASE, 2008.
[13] Kiriansky,V.,Bruening,D.,and Amarasinghe,S.Secure execution via program shepherding. In USENIX *Security*(Aug.2002).
[14] J. H. Perkins, M. D. Ernst, Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants, In *SIGSOFT'04/FSE12.*
[15] S. Hangal, S. narayanan, N. Chandra, S. Chakravorty, IODINE: A Tool to Automatically Infer Dynamic Invariants for Hardware Designs. In DAC 2005.

AUTHORS PROFILE

Mr. Partha Pratim Ray received B.Tech from the West Bengal University of Technology, India, in 2008. He is currently a final year student of M.Tech in the same university. His specialization is the Embedded Systems. The Author is also the student member of IEEE.

Dr. Ansuman Banerjee is an Assistant Professor at the Advanced Computing and Microelectronics Unit, Indian Statistical Institute Kolkata, India. He received his B.E. from Jadavpur University, and M.S. and Ph.D. degrees from the Indian Institute of Technology Kharagpur -- all in Computer Science. Prior to joining Indian Statistical Institute, he spent some time at the National University of Singapore as a research fellow and about three years at Interra Systems India Pvt. Ltd.

Mr. Banibrata Bag received B.Tech from the Burdwan University, India, in 2004. He mastered in VLSI and Microelctronics from the West Bengal university of Technology, India, in 2009. This author has more than four years experience as a software developer in PHP programming He is currently an Assistant Professor in department of ECE in Haldia Institute of Technology.