RADAR: A Tool for Debugging Regression Problems in C/C++ Software

Fabrizio Pastore[‡] and Leonardo Mariani[‡] and Alberto Goffi[⋆]
[‡]University of Milano - Bicocca , [⋆]University of Lugano
Email: {pastore,mariani}@disco.unimib.it , {alberto.goffi}@usi.ch

***Abstract*—Multiple tools can assist developers when debugging programs, but only a few solutions specifically target the common case of regression failures, to provide a more focused and effective support to debugging.**

In this paper we present RADAR, a tool that combines change identification and dynamic analysis to automatically explain regression problems with a list of suspicious differences in the behavior of the base and upgraded version of a program. The output produced by the tool is particularly beneficial to understand why an application failed.

A demo video is available at http://www.lta.disco.unimib.it/tools/radar/icse2013.html

I. Introduction

Regression problems occur when upgrades introduce faults in correctly working code. Since regression problems are known to be particularly popular and annoying, it is important to quickly identify and fix them.

Although several techniques can be used to assist developers when debugging programs, only few solutions have been tailored to effectively address regression problems. For instance well-known debugging techniques, such as Tarantula [1] and Delta Debugging [2], do not take advantage of the specific information available in case of regression problems. The additional information usually consists on a set of test cases that pass when executed on the base version of the software and fail when executed on the upgraded version of the same program. The availability of these test cases is relevant because the causes of the failure are hidden in the differences between the behavior of the base and upgraded version of the code when these test cases are executed.

A few techniques have been tailored to the case of regression problems. Recently, Yu et al. used Delta Debugging to identify the faulty instructions responsible for a regression fault by automatically generating and testing program versions that include increasingly smaller subsets of changes extracted from the upgrade under analysis [3]. Other techniques instead of simply pinpointing faulty instructions combine dynamic tracing with symbolic execution to characterize a regression fault in terms of differences between invalid and valid behaviors. For example, Darwin identifies legal program inputs that generate traces that are similar to the trace collected during a regression failure [4]. Golden instead characterizes the function with the regression fault in terms of differences in the weakest precondition extracted from the base and upgraded program [5].

Fault localization techniques are partially useful to developers because the identification of the suspicious instructions without additional contextual information does not concretely help software developers in understanding why an instruction is faulty and how to fix it [6]. The approaches that focus on the analysis of the faulty and legal behaviors produce more valuable information, but are known to not scale well because of the limitation of symbolic execution.

In this paper we present RADAR, a tool that helps software developers in debugging regression problems in C/C++ software. RADAR automatically identifies the behavioral differences between the failure, observed in the upgraded program, and the legal executions, observed in the base version of the program. RADAR does not localize faults but identifies a set of suspicious behaviors that can effectively drive the debugging activity toward the faulty areas of the upgrade. RADAR relies on lightweight static and dynamic analyses that make the approach scalable to large programs. Empirical results with open-source and industrial software systems are an early evidence of the effectiveness of the approach [7].

The paper is organized as follows. Section II presents a typical case that can be addressed with RADAR. Section III overviews the RADAR technology. Sections IV and V present the RADAR tool and illustrate how RADAR can be used by developers, respectively. Section VI reports some empirical results obtained with the RADAR tool. Section VII provides final remarks.

II. RADAR in Action: A Sample Scenario

To illustrate when and how RADAR can be used we present a sample scenario. To keep the sample scenario understandable we refer to a simple regression fault, anyway the scenario does not change significantly for complex faults.

Tom is the maintainer of *Accounting*, an application that manages the salaries of the workers in a company. A customer recently contacted the support service indicating the presence of a bug in version 1.0 of the application. After carefully inspecting the code Tom discovered that the problem is caused by method *getSalary*, which returns the salary of a worker given her ID. Method *getSalary* returns *NULL* when the ID provided as input does not belong to any active worker. This behavior is responsible for the failure because callers cannot distinguish the case of a worker with no salary from the case of a worker who left the company (remind that *NULL* corresponds to 0 in C++). To fix the bug, Tom introduces a

new case: *getSalary* returns -1 when a non-worker is passed as argument of the method (the fix is introduced in line 23 of Figure 1, by replacing the instruction `return NULL` with `return -1`).

```
20long getSalary(string ids){
21
22  if ( ! isWorker(id) ){
23    return NULL;
24  }
25
26  return workers.find( id)
      ->second;
27}
```

```
42 long getAverageSalary(list personIds){
43   list::iterator i;
44
45   long totalSalary = 0;
46   int workers = 0;
47
48   for(i=personIds.begin();
       i!=personIds.end();++i){
49     long salary = getSalary(*i)
50     if (salary == 0){
51       continue;
52     }
53     totalSalary += getSalary(*i);
54     workers++;
55   }
56   if (workers==0) return -1;
57   return totalSalary/workers;
58 }
```

Fig. 1. Methods *getSalary* (before the upgrade) and *getAverageSalary*.

To avoid the introduction of regression faults in the released software, Tom executes the available regression test suite and discovers the existence of a regression problem in the *getAverageSalary* function. Understanding why the computation is incorrect is not easy for Tom because he did not develop the *getAverageSalary* function. To ease the analysis of the regression problem, Tom runs RADAR.

RADAR automatically returned the chain of unexpected events that produced the failure. In particular RADAR shows that (1) during the failing execution the value of variable *salary* in line 50 is unexpectedly equals to *-1*, while in legal executions it was greater or equal than zero; (2) after the execution of line 55 variable *totalSalary* is unexpectedly decreased by one; and (3) the erroneous value of variable *totalSalary* is propagated through the loop thus leading to a wrong result for method *testGetAverageSalary*. It is clear that the case of the return value of function *getSalary* equals to *-1* is not supported by the function *getAverageSalary* and the rest of the anomalous behaviors are a consequence of the unsupported case. Tom can now successfully fix the function by adding a proper support to the missing case.

III. Debugging Regression Problems with RADAR

RADAR works in three steps: (1) generation of monitoring scripts, (2) data collection, (3) analysis. The following paragraphs describe each step.

A. Step 1: Generation of Monitoring Scripts

In the first step, the software developer specifies the paths to the executables and source folders of the base and upgraded versions of the program. This information is analyzed by RADAR to generate scripts that use the GDB debugger[1] to record data from program executions.

To keep monitoring lightweight, RADAR monitors only the program locations whose behavior is likely directly affected by the (erroneous) change: the modified functions (they may contain faults), their callers (erroneous return values may affect callers), and their callees (a fault may violate a function preconditions). The rest of the program locations that could be potentially affected by an erroneous change are ignored by RADAR. To collect data from comparable program locations, RADAR selects for monitoring only the statements that occur unaltered in both the base and upgraded version of a modified function.

RADAR identifies the modified lines of code by applying the Unix diff algorithm [8] to the source files. RADAR then uses objdump[2] to identify, from the executables, the functions that contain the modified code, their callers and callees.

A change may affect the behavior of a program altering both the sequence of operations executed by a function and the values assigned to variables. To capture these differences RADAR traces both every statement executed by a monitored function and the values assigned to variables.

B. Step 2: Data Collection

In the second step the software developer collects behavioral data using the GDB scripts automatically generated by RADAR during step 1. In this step, RADAR first executes the test cases for the base version of the program. Then RADAR executes the passing and failing tests of the upgraded version of the program, separately. RADAR automatically distinguishes passing and failing test cases on the basis of the return code of the executed program. During these executions RADAR traces the values of the variables and the sequences of executed statements.

C. Step 3: Analysis

In the third step RADAR analyzes the collected data and produces a report. The analysis starts with the generation of models that generalize and represent in a compact way the behavior of the base program when the execution terminates correctly. Models are generated from traces and could consist of Boolean expressions, which indicate the values that have been assigned to program variables during passing tests, and Finite State Automata (FSAs), which specify the sequences of statements that the program executes when the functions selected for monitoring are invoked.

RADAR derives Boolean properties from traces using Daikon [9]. Since RADAR monitors the body and the entry/exit points of functions, it can derive: *function preconditions*, that is properties that hold before the execution of a function, for instance for the case in Figure 1 RADAR detects that `this.workers != NULL` is a precondition of method *getSalary* (i.e., *WorkersMap.workers* is always initialized when *getSalary* is executed); *program properties*, that is properties that hold before the execution of a line of code, for instance for the case in Figure 1 RADAR detects that before the execution of line 50 in method *getAverageSalary* the following property holds: `salary >= 0`; *function postconditions*, that is properties that hold after the execution of a

[1] http://sources.redhat.com/gdb/

[2] http://www.gnu.org/software/binutils/

function, for instance for the case in Figure 1 RADAR detects that `RETURNVALUE >= 0` is a post-condition.

RADAR uses the KBehavior inference engine [10] to derive FSAs that generalize the sequences of operations executed by each monitored function. In this context an operation is the execution of a statement. KBehavior is effective in capturing the precedence relation among subsets of K events. This feature is particularly useful for RADAR, in fact changes often introduce issues in the precedence between events.

RADAR finally compares the models obtained by monitoring the base version of the program with the data recorded during the execution of the upgraded version of the software. Any event that occurs in the traces and is not accepted by models is classified as an anomaly. The anomalies identified uniquely when comparing the models with the traces collected from failing executions are reported to the user, while the anomalies identified also when comparing the models with successful executions of the upgraded program are classified as false positives, and thus they are filtered out.

The result of the comparison may be affected by structural changes like the addition or removal of methods and instructions. Simple changes like the addition of few lines of code in a function would change the line number of many other statements, and thus potentially invalidating the models that refer to program statements using line numbers. RADAR can accommodate many of these changes by automatically adapting the model to the upgraded software. See [7] for details.

IV. The RADAR Eclipse plug-in

We implemented RADAR as an Eclipse plug-in, which can be downloaded from http://www.lta.disco.unimib.it/tools/radar/.

Figure 2 shows the Eclipse workbench, which consists of multiple editors and views available within the Eclipse IDE: (a) the analysis view, (b) the trace editor, (c) the program points view, (d) the anomaly editor, and (e) the customized properties view.

The *analysis view* shows the data generated by RADAR during the analysis of a particular regression problem: it lists the execution traces that have been recorded (RADAR shows a single trace for each thread of the application), the models associated to each monitored program-point, and the results of the analysis.

The *trace editor* shows the sequence of statements executed by the program, limitedly to the statements that have been selected for monitoring. The *program points view* shows the values of the variables that are in the scope of the execution for a program point selected from the trace. For instance, Figure 2-d shows the variable values collected when executing line 50 in function *getAverageSalary*.

The *anomaly editor* shows the anomalies returned by RADAR, the program locations that generate them, and the models that detected the anomalies. The *properties view* shows additional information about a selected anomaly, such as the anomalous values of the variables and the content of the stack trace. The editor provides a contextual menu for opening the trace that includes the anomalous event, the model that detected an anomaly, and the source program with the line of code that generated a selected anomaly.

V. Debugging Regression Problems with RADAR

The software developer who needs to debug a regression fault with RADAR has to complete three activities: (1) create a new analysis configuration, (2) monitor the execution of the test suites, and (3) run the analysis and inspect the results.

Developers create a *new analysis configuration* by using a wizard that guides the user toward setting the paths to the source folders and the executables of the base and upgraded versions of the software. Although RADAR automatically identifies the functions that must be monitored according to the changed area of the code, it is also possible to manually modify the list of functions and methods that must be monitored. In this way the developer can exploit a knowledge specific to the system under analysis, for instance by including functions of critical relevance that are not selected by the tool or excluding functions that are known to be irrelevant. The specification of the functions that must be monitored can be performed using a syntax based on regular expressions.

Test cases for C/C++ software are often executed either from the command prompt or through shell scripts. RADAR assists the developers by printing the shell commands that should be executed to run test cases and monitor the application. This step is usually quite simple. For instance, the following commands must be executed to monitor the example application introduced in Section II:

```
CONFIG=/home/Tom/WorkersMap/original.gdb.txt
GDB="gdb -batch -silent -n -x $CONFIG --args"
$GDB WorkersMapTest testNoWorker test1Worker ...
```

These lines are printed by RADAR. The first and second line are executed to set some environment variables. The third line executes the tests through GDB.

In the last step, developers can run the analysis of the traces using a contextual menu. When the analysis is complete, RADAR opens an editor with the list of detected anomalies. Software developers inspect the anomalies in order of appearance. In case the anomaly is produced by a data property, developers visualize the variables values that caused the anomaly. In case the anomaly is produced by a FSA, developers can open the FSA to determine the expected operations.

For instance, Figure 2 shows the set of anomalies returned in the scenario illustrated in Section II. The first line of the output indicates that before the execution of line 50 the value of variable `salary` is not accepted by the Boolean expression `salary >= 0`. Thus in the upgraded version of the program, variable `salary` is unexpectedly lower than 0: its value is equal to -1 as shown by the properties view in Figure 2-e.

A video demonstrating the use of RADAR is available at http://www.lta.disco.unimib.it/tools/radar/icse2013.html

VI. Empirical Results

We have used RADAR to debug 7 regression faults affecting open source software systems and 3 regression faults injected

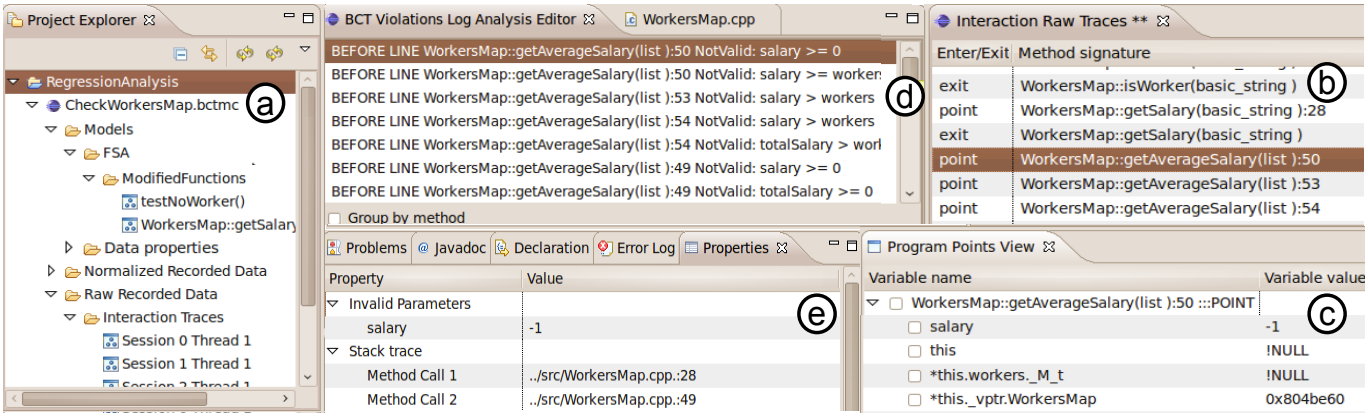

Fig. 2. The Eclipse workbench augmented with the editors and views provide RADAR's views.

on an industrial system [7].

Results show that RADAR can effectively help software developers in debugging regression problems. In 8 of the 10 cases RADAR successfully spotted anomalies that capture the misbehavior that generated the failure. In the two unsuccessful cases RADAR was not able to monitor the target system because, instead of analyzing a regular change, we analyze two different releases, which include many pervasiveness changes that caused the monitoring of a large portion of the software (more than 500 monitoring breakpoints have been generated).

Although we had this issue in our experiment, we expect that this limitation will not affect the applicability of RADAR in most of the practical cases. Modern development processes in fact postulate the adoption of continuous integration mechanisms that execute regression test suites periodically (e.g., nightly), thus identifying regression problems after few changes are performed on the system.

The effort required for inspecting the anomalies reported by RADAR is low: on average RADAR identified 4.25 different anomalies for each case study. Results are characterized by a pretty high precision: 70% of the reported anomalies spot misbehaviors that depend on the fault. Both the Boolean properties and FSA models resulted to be useful to capture relevant anomalies: in 2 cases anomalies are discovered by FSAs only, in 3 cases by Boolean properties only, and in the remaining cases by both. To give an intuition of the amount of effort required to debug software with RADAR, we computed the average number of instructions that separate the line of code with the fault from the closest line that generated a true anomaly. This distance amount to 3.75, which indicates that RADAR not only provides a valid description of the faulty behavior in terms of anomalous events, but can also drive the developers close to the faulty statements. Additional details about our experiments are available in [7].

VII. Conclusion

In this paper we presented RADAR, a tool that helps software developers in debugging regression faults. RADAR automatically spots the behavioral differences between the legal executions in the base version of the software and a failing execution in the upgraded version of the software. RADAR derives models that generalize the data observed in legal executions, and uses these models to identify the anomalous data values and the unexpected sequences of operations that caused a regression failure.

Early empirical results with open source and industrial systems suggest that RADAR can be an effective tool supporting developers.

Acknowledgment

This work is partially supported by the European Community under the call FP7-ICT-2009-5 project PINCETTE 257647.

References

[1] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the International Conference on Automated Software Engineering*, 2005.

[2] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *Proceedings of the European Software Engineering Conference held jointly with the International Symposium on Foundations of Software Engineering*, 1999.

[3] K. Yu, M. Lin, J. Chen, and X. Zhang, "Practical isolation of failure-inducing changes for debugging regression faults," in *Proceedings of the International Conference on Automated Software Engineering*, 2012.

[4] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani, "Darwin: An approach to debugging evolving programs," *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 3, pp. 1–29, Jul. 2012.

[5] A. Banerjee, A. Roychoudhury, J. A. Harlie, and Z. Liang, "Golden implementation driven software debugging," in *Proceedings of the international Symposium on Foundations of Software Engineering*, 2010.

[6] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the International Symposium on Software Testing and Analysis*, 2011.

[7] F. Pastore, L. Mariani, A. Goffi, M. Oriol, and M. Wahler, "Dynamic analysis of upgrades in c/c++ software," in *Proceedings of the International Symposium on Software Reliability Engineering*, 2012.

[8] W. Miller and E. Myers, "A file comparison program," *Software-Practice and Experience*, vol. 15, no. 11, pp. 1025–1040, 1985.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[10] L. Mariani, F. Pastore, and M. Pezze, "Dynamic analysis for diagnosing integration faults," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 486–508, 2011.