Dynamic Analysis of Upgrades in C/C++ Software

Fabrizio Pastore, Leonardo Mariani and Alberto Goffi
Department of Informatics, Systems and Communication
University of Milano Bicocca, Milan, Italy
{*pastore, mariani, goffi*}*@disco.unimib.it*

Manuel Oriol and Michael Wahler
ABB Corporate Research, Industrial Software Systems
Baden-Dättwil, Switzerland
{*manuel.oriol, michael.wahler*}*@ch.abb.com*

***Abstract*—Regression testing techniques are commonly used to validate the correctness of upgrades. When a regression test fails, testers must understand the erroneous behaviors that caused the failure and identify the fault that originated these erroneous behaviors. In many cases, identifying the causes of a failure is difficult and time consuming. The analysis of regression problems provides interesting opportunities to validation and verification techniques. In fact, by comparing the execution of the base version and the upgraded version of the same program it is possible to automatically deduce information about incorrect behavior of the program.**

In this paper we present RADAR (Regression Analysis with Diff And Recording), a dynamic analysis technique, which analyzes regression problems and automatically identifies the chain of erroneous events that lead to a failure in C/C++ programs. RADAR exploits information about changes and the availability of multiple versions of the same program to automatically distinguish correct and suspicious events. Empirical experience with industrial and open source cases shows that RADAR can effectively support testers in the investigation of regression problems. Thus, RADAR can drive and simplify the debugging process.

***Keywords*-dynamic analysis, upgrades, regression testing, debugging, model inference**

I. Introduction

Software systems are constantly modified by software engineers to satisfy new requirements, support new technologies, and repair bugs. Frequent changes to a software system, although often necessary, might represent a threat to its integrity. To mitigate this risk, software engineers apply quality control techniques as diverse as regression testing or formal methods.

Regression testing techniques, for example, are commonly used to check the correctness of changes and make sure that changes do not negatively affect the unchanged code. A number of different strategies identify [1], [2] and prioritize [3], [4], [5] the test cases to execute when validating a change.

When these techniques find a failure, accurately identifying its cause allows programmers to fix it in an efficient way. Multiple techniques can be used to analyze failed executions and detect the erroneous events that led to a failure [6], [7], [8], [9]. None of them are however specifically tailored to the analysis of regression failures — failures resulting from version changes.

Regression problems simplify failure analysis in two ways:

- *Space*: since the functionality that does not work in the upgraded version works correctly in the base version, the problem has been introduced by the change. The analysis could thus be focused on the modified instructions of the software.
- *Behavior*: the execution that fails in the upgraded version of the program can be compared with the same (correct) execution in the base version of the program. Identifying the suspicious events occurring in one case but not in the other could thus ease debugging.

A technique that analyzes regression failures and takes advantage of these assumptions has a chance of being highly effective. Note that testers clearly know that they are analyzing a regression problem because by definition, a regression problem is evidenced by a test case passing for the base version of the program and failing for the upgraded version. In this case, testers also know that they should use a technique specific to the analysis of regression failures.

This paper presents RADAR, a dynamic analysis technique, which analyzes regression problems and automatically identifies failure causes in C and C++ software. RADAR combines change analysis, monitoring, model inference, and anomaly detection to identify the key events responsible for a failure and present them to testers with the aim of easing the debugging process. Testers can follow the chain of suspicious events identified by RADAR to trace the failure back to the erroneous conditions that determined the failure and thus, the fault that led to the failure.

The major contributions of this paper are:

- a change analysis technique that identifies the program locations that should be monitored to detect the events responsible for the failure;
- a weakly intrusive and change-specific monitoring solution that can be applied to any C/C++ program, regardless of the compiler used;
- an approach for the generation of models that represent the (legal) behavior of the application at strategic program points;
- a dynamic analysis solution that automatically identifies the suspicious events from the trace collected for the

failed execution;
- empirical evidences that RADAR can effectively address regression problems.

The paper is organized as follows. Section II presents the main steps of the technique. Section III introduces a running example used throughout the rest of the paper. Section IV details how RADAR extracts data from C/C++ applications. Section V describes the generation of models. Section VI presents the RADAR failure analysis. Section VII reports empirical results. Section VIII discusses related work and we conclude in Section IX.

II. RADAR

RADAR is a technique for automatically identifying the likely causes of regression problems in C/C++ software. We use the term regression problem to indicate a failure that depends on a fault that (1) is introduced as a consequence of an upgrade (namely from a base to an upgraded version), and (2) affects an existing functionality. There should thus be a test case expected to pass for both versions of a program, but passes only for the base version and fails when executed on the upgraded version.

The idea underlying RADAR is that the causes of a failure produced by a regression fault can be identified by looking at the differences between the correct behavior, shown by the base version, and the erroneous behavior, shown by the upgraded version. Since the problem affects a functionality that exists in both the base and upgraded versions of the program, the causes of the faulty behavior must necessarily be related to events that do not occur in the base version of the program, but only occur in the upgraded program. To compare the failing execution with the behavior of the base version of the program, RADAR derives multiple models that represent the behavior of the base version of the program, and then compares the failed execution with these models to identify a chain of anomalous events that can explain the failure. Testers can follow the chain of anomalous events to understand the failure and reach the fault location. RADAR compares failed executions with models because models can successfully represent the legal sequences of events produced in many valid executions. The behavior represented in the models is not representative of a single case, (e.g., a single passing test case), but it is representative of every passing test that has been executed. This characteristic makes models particularly suitable to be used as anomaly detectors for upgrades. In fact, since models represent the general correct behavior of the base version of a system, they tend to generate many useful alarms (i.e., illegal events generated by the upgraded program) and only few false alarms (i.e., legal events that can be generated in the upgraded program but could never be generated by the base version of the software), when used to check an execution that fails in the upgraded version of the program.

In opposition to classic fault localization techniques [10], [11], [12], which guess the fault location without providing any explanation of the guessing, RADAR aims at discovering why the application failed, leaving to the tester the due of identifying the location of the fault. The rationale is that the likely fault location alone is a relatively useful information because the tester must still analyze and understand the failed execution to confirm the presence of the fault, and in many cases the fault is not in the locations that are inspected first. RADAR provides information that is complemental to the fault location and that can facilitate debugging.

RADAR analyzes regression faults in three phases, as shown in Figure 1. In the first phase, *Script Generation*, RADAR identifies the program locations that must be used as observation points for identifying the causes of a failure (*Change Analysis* step). Since the fault is introduced during an upgrade, the locations that most promisingly can produce information about the anomalous events are the program locations in the neighborhood of the changes. The changed locations themselves are excluded because they potentially differ in significant ways. The considered locations are the same in both the upgraded and base versions of the program and the behavior of the two program versions at these points can be easily compared (e.g., since the two statements are the same, they include the same variables, the same function calls, etc.).

RADAR monitors C and C++ programs using a program debugger: GDB[1]. RADAR automatically generates the GDB scripts later used to monitor the base and upgraded versions of the program under analysis (*Script Creation* step).

In the second phase, *Model Generation*, the base version of the application is executed through GDB using the test suite of the application. The script generated in the first phase collects data about the behavior of the program in term of values assigned to program variables, executed statements and functions and method calls originated by the changed functions (*Test Suite Execution* step). The monitored data is recorded into trace files and is then used to distill models. These models generalize and capture in a compact way the values that are assigned to variables and the event flow produced by the program in correct executions (*Model Inference* step). RADAR uses two models: Boolean properties, to represent the values that can be assigned to variables, and Finite State Automata, to represent the sequences of computational steps (i.e., the statements) and method/function calls executed by a function or a method.

In the third phase, namely *Failure Analysis*, the test case that causes the failure on the upgraded application is executed, and the behavior of the program is recorded in a trace file (*Test Case Execution* step). The trace is then compared with the models distilled from the base version of the program to look for anomalous behaviors, that is

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

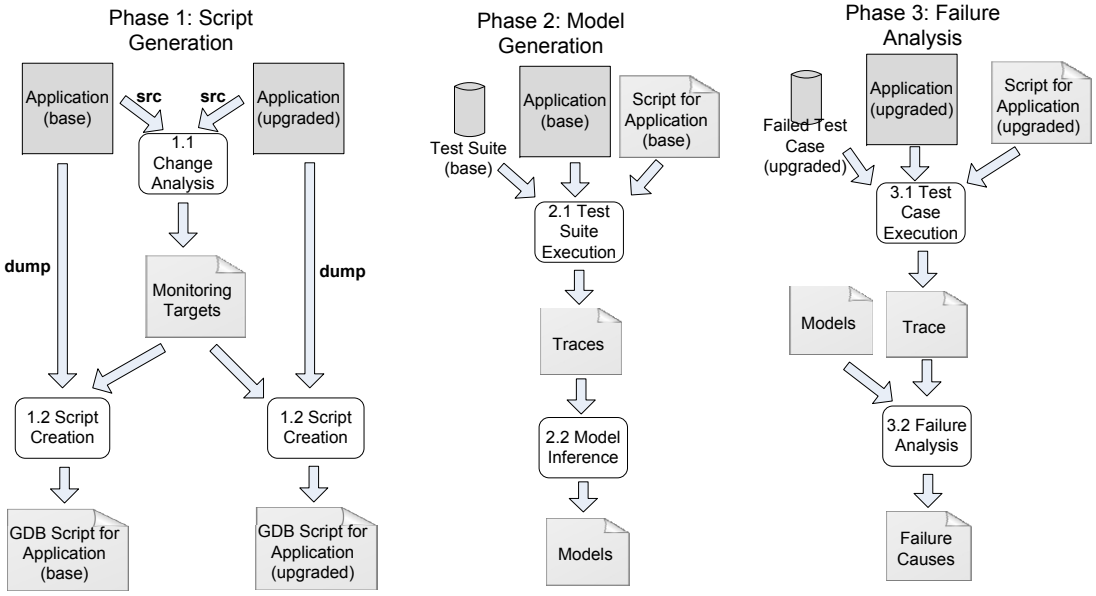

Figure 1. The RADAR approach.

values of variables, computational steps or function calls that are not accepted by the corresponding model (*Failure Analysis* step). In this phase, specific heuristics can be used to remove false alarms from the set of anomalies that have been identified. The chain of anomalous events discovered in this phase is presented to the tester to explain the likely sequences of events that led to the failure of the application.

In the next sections we describe in detail how each phase works, with the support of a running example.

III. Running Example

In this section we introduce a simple running example that we use to illustrate the technique. The example represents the case of a change that introduces a regression problem in a small application that computes the salaries of the workers in a company. For the purpose of the example we concentrate on two functions: `getAverageSalary`, which computes the average salary for the workers in the list passed as parameter, and `getSalary`, which is an auxiliary function that returns the salary of the worker specified as parameter. The code of the two functions is shown in Listing 1.

The `getAverageSalary` function computes the average salary considering various types of workers. In particular, the list passed to `getAverageSalary` may include people who are not anymore workers of the company (e.g., laid off and retired workers) that must not be taken into account in the computation of the average salary. To discard these workers from the computation, the `getSalary` function checks if the specified worker is still an active worker by invoking the function `isWorker`, not shown in the listing. If a specified worker is not an active worker, the `getSalary` returns 0. This value is intercepted by the `getAverageSalary` function, which discards from the computation a worker with a 0 salary (see line 49).

Listing 1. Compute Average Salary Example

```
42 long WorkersMap::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       if (getSalary(*i)==0) continue;
50
51       totalSalary += getSalary(*i);
52       workers++;
53    }
54
55    if (workers==0) return -1;
56
57    return totalSalary/workers;
58 }
59
60 long WorkersMap::getSalary(string workerId){
61
62    if (!isWorker(workerId)) {
63       return 0; //upgraded with return -1;
64    }
65
66    return workers.find(workerId)->second;
67 }
```

For the purpose of the example, we consider a change consisting in the introduction of the capability of handling additional types of workers. In particular, some workers can have a salary equals to 0 (e.g., students with reimbursement for expenses, but no salary) and the `getSalary` function must be refined to distinguish non-workers from workers with a 0 salary. The new version of the `getSalary` function returns -1 instead of 0 for non-workers (see line 63 in the example, the comment shows the statement introduced

in the upgraded version of the program). We assume that developers miss the change that must be implemented at the caller side, and the `getAverageSalary` function still checks for 0 values instead of checking for the return value equals to -1. This is unlikely to happen for such a small example, but when the changed function is part of a large application and it is invoked in many places this type of fault is quite frequent.

The fault introduced with the upgrade results in the generation of incorrect average salaries when non-workers are included in the list passed as parameter of the function `getAverageSalary`. For example the regression test shown in the Listing 2 passes for the base version of the program, but exposes the failure in the upgraded version. The average salary expected by the assertion is 30,000 while the value returned by the upgraded function is 19,999. The problem is revealed by the presence of `worker1`, which is not a worker, in the list.

Listing 2. Regression Test Example

```
void testGetAverageSalary_bug(){

 WorkersMap map;

 string worker1("MRTFRZ83D6YETZD");

 string worker2("PSTFRZ83D6YETZD");
 map.addWorker(worker2, 50000);

 string worker3("PBTFRZ83D6YETZD");
 map.addWorker(worker3, 10000);

 list<string> workers;
 workers.push_back(worker1);
 workers.push_back(worker2);
 workers.push_back(worker3);

 assertEquals((50000+10000)/2,
            map.getAverageSalary(workers));
}
```

In the next sections, we show how RADAR can be used to analyze this regression problem.

IV. Script Generation

In the Script Generation phase, RADAR generates monitoring scripts in two steps. In the first step, RADAR identifies the monitoring targets. In the second step, RADAR generates the scripts that collect the data for the analysis.

To identify the monitoring targets, RADAR relies on two facts. First, the data collected by executing the base and upgraded versions of a same program can be directly compared only if the data refer to a same (unchanged) program location that exists in both versions. In fact, the data collected from changed code locations may refer to completely different variables and modified algorithmic steps, and thus be incomparable. Code regions that have not been modified are expected to have a similar behavior across versions. Second, to detect the (negative) impact of a change, it is intuitively more effective to monitor the unchanged areas of code that occur close to the changed ones[2] rather than other areas. In fact, these areas use comparable variables and produce results that are likely influenced by the change, and can thus be used to recognize anomalous events.

To identify the code locations that have been changed, RADAR compares program versions using diff[3]. In our first version of the approach we found diff effective enough, despite its simplicity. In the future, other ways of detecting changes could be experienced. Once the changed lines of code have been identified, RADAR selects every function/method that includes at least a modified line, and every function/method that invokes at least a modified function/method. For each selected function/method, RADAR classifies its entry/exit points and every unchanged line of code in its body as a monitoring target. The monitoring targets are the observation points used by RADAR to detect the anomalous events. To effectively monitor programs at runtime RADAR generates monitoring scripts that, for each program point, allow to: (1) trace the execution of the statement, (2) record the value of every local and global variable valid in the scope of the line, (3) record the sequence of method/function calls originated by the execution of each selected function/method call. If the application under analysis makes extensive use of standard libraries that produce an unmanageable and noisy number of function calls, the testers can disable the monitoring of the libraries and record only the interactions between entities defined in the application.

If we consider our running example, a single line of code has been modified (line 63). The functions selected by RADAR are both `getSalary`, because it includes a changed line of code, and `getAverageSalary`, because it directly invokes a function that has been modified. The monitoring targets are the entry and exit points of the `getSalary` and `getAverageSalary` functions, plus every line of code included in the `getSalary` and `getAverageSalary` functions, with the exception of line 63. For each program location identified as monitoring target, RADAR records the value of local and global variables. In addition RADAR traces the sequences of function/method calls originated by the `getSalary` and `getAverageSalary` functions.

Note that even if the two functions `getSalary` and `getAverageSalary` would be embedded in a larger project, RADAR would anyway automatically restrict monitoring to the `getSalary` and `getAverageSalary` functions only, unless there exist other functions directly invoking `getSalary`. The ability of focusing the monitoring to the code that is correlated to the change is a unique ability of RADAR. Other dynamic analysis techniques sample the executions with a granularity that is independent from the change under analysis, thus losing

[2]Distance between code locations can be measured as the number of nodes that separate the locations in the control-flow.

[3]http://linux.about.com/library/cmd/blcmdl1_diff.htm

a lot of information that would simplify the understanding of the failure. For example, multiple techniques focus on function entry/exit points only, without considering the body of the functions [9], [13]. If we consider the running example, these techniques could detect the new value that the function `getSalary` can return, but this is an expected anomaly, and would fail in detecting the many erroneous results produced by the computation from line 49 to line 52 of function `getAverageSalary`, which can be detected with RADAR.

To collect data at run-time, RADAR synthesizes GDB scripts that run the target program, interrupt the execution when a monitoring target is reached and collect the data useful to RADAR. The use of GDB for monitoring satisfies multiple requirements that are extremely important for C/C++ software:

- **Applicable to most of the compilers** A number of compilers are available for C/C++ programs, and different organizations take different decisions about the compiler that better adapts to the their projects. Unless restricting the solution to a core of the language, it is difficult to find a monitoring approach that works for every compiler. The use of GDB produces a monitoring solution that is applicable to any compiler that can produce object code that includes debugging options, which consist of the majority of the compilers;
- **Weakly intrusive** Monitoring through GDB does not require the modification of the source code and impacts on the object code through the standardized process, implemented in most compilers, of enclosing debugging symbols in the object code [14].
- **Fine grained monitoring** Using GDB scripts the monitoring can take place at every statement, while many popular monitoring approaches do not support this granularity;
- **Extensive access to data variables** Through GDB scripts it is extremely simple to access every variable that is alive and in the scope of the program location where the execution is interrupted. On the contrary, using other monitoring approaches it is extremely hard or even impossible to monitor non-local variables.

According to our solution the approach based on GDB works well when the program does not have to satisfy real-time requirements or when data are not collected to check the correctness of real-time behaviors. In fact, monitoring through GDB is rather expensive and real-time behaviors might be negatively affected by the approach.

Following paragraph shows an excerpt of the script that collects the data for the line of code 52 in the `getAverageSalary` function. Note that the symbol ... in the script indicates removed text.

```
# code for line POINT getAverageSalary(...):52
b /home/.../src/WorkersMap.cpp:52
commands
silent
echo !!!BCT-POINT: getAverageSalary(...):52\n
echo !!!BCT-stack\n
bt
echo !!!BCT-stack-end\n
echo !!!BCT-args\n
info args
echo !!!BCT-args-end\n
echo !!!BCT-locals\n
info locals
bct_print_pointers this
echo \n
echo !!!BCT-locals-end\n
c
end
```

In a nutshell the script sets a breakpoint at line 52 (command `b`) and specifies a set of commands that must be executed before continuing the execution. The set of commands starts with the command `commands` and ends with the command `end`. The commands starting with `echo` produce outputs that facilitate the parsing of the results. The most important commands that are executed are: `bt`, to record the content of the stack; `info args` to record the values of the parameters, if any; `info locals` to record the values of the local variables; `bct_print_pointers this` to record the values of the fields, if the execution is stopped in a method; and `c` to continue the execution.

In this specific example no global variables are recorded. Since at runtime there exist a huge number of global variables, and recording all of them would be too expensive, RADAR heuristically records only the global variables that occur at least once in the source file that includes the code with the breakpoint. Intuitively, global variables that are not used in the file that includes the code under analysis are likely to be irrelevant for the execution of the monitored code. If no global variable occurs in the file, like in the running example, the command for recording global variables is not even included in the script.

V. Model Generation

In the model generation phase, RADAR produces models that represent in a general and compact way the behavior of the base version of the program. In particular, RADAR derives two types of models. Models that represent the values that can be assigned to variables at each monitored program point, and models that represent the sequence of steps and calls that can be executed by a function.

To derive a model for the values that can be assigned to variables, RADAR runs the Daikon inference engine [15] on the data collected at each program point. The output produced by Daikon consists of a set of Boolean expressions that hold at the corresponding program point. For instance, some of the models obtained with Daikon for the running example are shown in the following lines.

```
MODELS THAT HOLD AT LINE 46
personIds != null
totalSalary == 0
workers != 0
...

MODELS THAT HOLD AT LINE 49
totalSalary >= 0
workers >= 0
i != null
...

MODELS THAT HOLD AT LINE 66
workerId != null
this != null
...
```

To derive a model that represents the computations and calls executed by the monitored functions RADAR uses kBehavior [9]. kBehavior accepts in input a set of traces and returns an automaton that generalizes the behavior represented in the traces (the automaton is guaranteed to accept every trace provided as input). In our case, each function/method selected for monitoring produces a set of traces. Each trace associated with a function/method contains a sequence of line numbers, which represent the (unchanged) lines of code that are executed by a test, interleaved with function names, representing the function calls produced by the monitored function when executing a test. For instance, the automaton obtained with kBehavior from a sample set of traces for the `getAverageSalary` function is shown in Figure 2. The model represents the sequences of operations, including method calls, that are legal according to the available test suite.

Line numbers and method/function calls used in the model might be an issue for the analysis of the upgraded program: an upgrade might alter line numbers and might add/remove functions. RADAR includes mechanisms to handle these cases. In particular, RADAR exploits the output of diff, which identified the corresponding program statements in phase 1 of the technique, to automatically convert the line numbers of the base version into the corresponding line numbers of the upgraded version. Moreover, monitoring works in a specific way when a removed or added function is executed. When a removed function is executed on the base version of a program, the corresponding trace will not report the call to the removed function, which would not exist in the upgraded program, but will include the calls originated from the body of the removed function, if any. Similarly, when a new function is executed on the upgraded version of the program, the corresponding trace will not report the call to the new function, which would not exist in the model, but will include the calls originated from the body of the new function, if any. By using these mechanisms automata can be safely reused to analyze upgrades.

In summary, Boolean expressions and automata capture the values that can be assigned to variables and the computational steps that can be executed by functions when the execution terminates correctly, respectively. These models are used to detect anomalous values in the trace recorded from the failed execution in the upgraded version of the program.

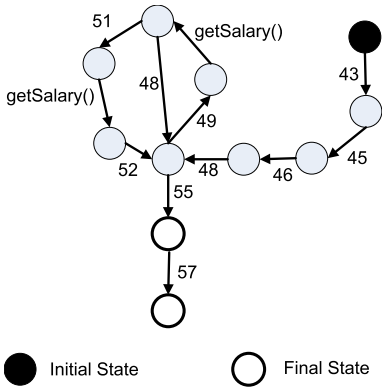

Figure 2. The FSA for the `getAverageSalary` method.

VI. Failure Analysis

In the last phase, RADAR executes the test case that produces the failure, records the corresponding trace, and compares the trace with the model. The script for recording the data from the upgraded program is generated in the script generation phase. RADAR compares the content of the trace with the models. In particular, it compares the values of the variables in the failed execution with the models on the program variables, and it compares the computational steps and sequences of calls executed by the selected functions with the automata. RADAR analyzes failing test cases only, this way it ignores anomalies caused by behavioral changes that are already verified by passing test cases, and focusses only on the anomalies that could have caused the failure.

Every time a model is violated, an anomalous event is detected. A Boolean property that holds at a given program location is violated if in the failed execution the values of the variables at the corresponding program locations do not satisfy the property. An automaton is violated if it rejects the sequence of events (events include the execution of an unchanged line of code and the invocation of a function/method) executed in the failed execution.

To ease debugging, RADAR reports the detected anomalies in the order they are observed, that is from the earliest to the latest. The tester can follow the sequence of events to understand the erroneous events produced by the fault. In the case of the running example, the failed execution does not violate any automaton, but violates a number of properties on program variables. In the following we report the models that are violated respecting the temporal order in which they occur. The numbers in the left column indicate the ordering of the events. The name of the method/function followed by : and the line number indicate the program instruction that

is executed when the violation occurs. The keyword `EXIT` instead of a line number indicates an anomaly detected when exiting from the function/method.

```
1: getSalary:EXIT returnValue >= 0
2: getAverageSalary:52 totalSalary > workers
3: getAverageSalary:49 totalSalary >= 0
4: getAverageSalary:55 totalSalary oneof
                              {0, 60000, 118000}
5: getAverageSalary:55 workers oneof {0, 2, 6}
6: getAverageSalary:57 totalSalary oneof
                                 {60000, 118000}
7: getAverageSalary:57 workers oneof {2, 6}
8: getAverageSalary:EXIT returnValue oneof
                                {-1, 19666, 30000}
```

If we follow the chain of anomalies it is simple to understand the chain of events that produced the failure. In fact, RADAR detects that `getSalary` returned a negative value, which never occurred in the past. The fact that the failure is related to the new value that can be returned was expected because it was the only change we implemented. However, RADAR also detects that the new return value has an impact on the `getAverageSalary` function. This is extremely useful because the same function can be used in multiple places and we have an exact indication of where the erroneous events are generated. RADAR also indicates that the new return value impacts on the value of the `totalSalary` inside the `for` loop. In the correct executions `totalSalary` is always positive, while in the failed execution it has been also assigned with negative values. The erroneous computation of the `totalSalary` naturally impacts also on the value of `workers` and `totalSalary`, successively in the execution. This chain of events gives intuitions about the reason of the failure and the tester can easily detect that the value of `totalSalary` in `getAverageSalary` depends from the `if` statement at line 49, and that the condition must be fixed.

Note that the distribution of the anomalies is strictly dependent from the fault being analyzed. When the fault is related to the use of elements like API, objects, and libraries, the anomalies are likely to occur in the automata. Whereas if the data model underlying the application is modified, like in the running example, the anomalies are likely to occur on the variable values. In many non-trivial cases, the anomalies are of heterogeneous type.

When the complexity of the program increases, spurious model violations may occur, especially the ones related to program variables. These model violations complicate the understanding of the chain of events that lead to a failure and must be removed from the output. RADAR implements the following simple heuristic to mitigate the effect of spurious model violations: "if an anomaly detected at a program location `l` is produced by a model about a variable that never occurs in the function that includes `l`, the anomaly is removed".

RADAR has been useful in this extremely simple example, but it can be even more effective when the change and the complexity of the program are non-trivial. In the next section, we show how RADAR has been effective with multiple open-source and industrial C and C++ programs.

VII. Empirical Validation

To empirically validate RADAR we applied it to changes in both open source and industrial software. In the former case, we extracted the cases from the erroneous changes isolated by Yu et al. [16], and made available at http://code.google.com/p/ddexpr/. These cases consist of multiple erroneous changes that affected popular programs available in Linux distributions. In the latter case, we selected multiple upgrades of the FASA system, which is a real-time software framework for distributed control systems developed at ABB Corporate Research [17], and we manually injected faults that are semantically correlated to the considered change. Because of intellectual property reasons we cannot make the code of the industrial cases available.

We analyzed these cases with our RADAR implementation, which is available at http://www.lta.disco.unimib.it/tools/radar/. In total, we analyzed 10 upgrades.

Table I reports the empirical results for the 8 cases in which RADAR has been successful. We discuss at the end of this section the two unsuccessful cases. Column *Application* indicates the *Name* of the application, its *Lang*uage, which can be either C or C++, its *Type*, which can be *Open Source* or *Industrial*, and its size, expressed as lines of code (*LOC*). Column *Change* specifies the *Type* and the *Size* of the analyzed change. The type can be either *Extension*, which indicates an upgrade that adds functionalities to an application; *Bug Fix*, which indicates an upgrade that fixes a bug; or *Refactoring*, which indicates an upgrade aimed at improving the internal quality of the program. Column *Size* gives an intuition of the spreading of the changes by reporting the number of functions altered by the change. Column *Filtered Anomalies* indicates the number of anomalies filtered by the heuristic described in Section VI. Column *Reported Anomalies* provides details about the anomalies discovered by RADAR, restricted to the ones that passed the filtering step. In particular, we report the total number of anomalies (column *Tot*), the number of false positives (column *FP*), the number of true positives (column *TP*) distinguished between true positives discovered with Boolean properties (column *TP Bool*) and true positives discovered with automata (column *TP FSA*), and the *Precision* of the approach ($precision = \frac{TP}{TP+FP}$). A true positive is an anomaly that corresponds to an erroneous condition generated by the fault. A false positive is a spurious anomaly that does not correspond to any erroneous condition. In the case of the FASA system the software engineers from ABB evaluated and classified the anomalies. In the open source applications, the authors of the paper evaluated and classified anomalies. Precision indicates the rate of correct

Table I
EMPIRICAL RESULTS

Application				Change		Filtered	Reported Anomalies						Distance
Name	Lang	Type	LOC	Type	Size	Anomalies	Tot	FP	TP	TP Bool	TP FSA	Prec	
grep_a	C	OpenSource	22K	Extension	10 func.	6	7	2	5	1	4	0.71	0
grep_b	C	OpenSource	22K	Refactoring	2 func.	17	7	2	5	0	5	0.71	5
bc	C	OpenSource	10K	Extension	41 func.	0	4	2	2	1	1	0.5	12
find_a	C	OpenSource	24K	Bug Fix	4 func.	17	5	1	4	3	1	0.8	2
diff	C	OpenSource	10K	Bug Fix	1 func.	177	48	0	48	48	0	1	5
FASA-1	C++	Industrial	39K	Bug Fix	1 func.	0	2	0	2	2	0	1	2
FASA-2	C++	Industrial	38K	Refactoring	3 func.	0	5	3	2	2	0	0.4	2
FASA-3	C++	Industrial	39K	Refactoring	4 func.	0	2	1	1	0	1	0.5	2

and interesting warnings produced by the technique. Finally, column *Distance* indicates the distance measured as the number of lines of code that separate the first true positive detected by RADAR and the location of the fault according to the control-flow graph of the application.

According to the empirical results the filtering step demonstrated to be useful even if not always necessary. In fact, even if in half of the cases no anomalies have been filtered, in three cases more than 10 anomalies have been automatically removed, and in one case even 177 anomalies have been automatically removed, finally obtaining outputs that can be easily handled manually.

If we consider the anomalies that pass the filtering step, the data in Table I show that most of these anomalies are relevant anomalies that can be beneficial to failure understanding and fault localization. In fact, the precision of the approach (i.e., the density of the useful anomalies) ranges from 0.4 to 1.0, and in 63% of the cases is higher than 0.7. It is also worth mentioning that some faults generated anomalies of a same type (violations of Boolean expression or violations of automata), but multiple faults generated anomalies of both types, demonstrating that both are useful to understand failures.

In most of the cases RADAR kept the total number of returned anomalies small (the median is 5). Only in the *diff* case the number of anomalies is pretty high. However, in the *diff* case the anomalies can be easily handled by software testers because all the anomalies are generated by the same two Boolean properties that are violated in 25 different code locations of the modified method. Developers obtain information useful to understand the failure by simply inspecting two of the 48 anomalies.

We interpret the data about the good precision and the small number of generated anomalies as the capability of RADAR to produce a core set of important anomalies that can concisely explain the failure causes and can bootstrap the debugging process in the right direction.

RADAR has been also good in producing a chain of anomalous events that is rooted at the nearby of the fault location. In fact the median of the distance between the true positive closest to the fault location and the fault location is 2 statements. This information suggests that the targeted monitoring and run-time analysis implemented in RADAR is beneficial also in terms of the detection of fine-grained events that are close to the location of the fault.

The obtained results are even more relevant if interpreted as complementary to fault localization information. Yu et al. for example report that the delta debugging fault localization technique successfully identified the erroneous changes in five of the seven open source regression faults considered in this paper (*bc*, *grep_a*, *grep_a*, *find_b*, *indent*), but only in the case of *bc* software developers can easily repair the software just by reading the lines reported as erroneous. In the other cases they require an in depth understanding of the software behavior. If the information about the fault location returned by a fault localization technique is combined with the anomalies returned by RADAR the tester would gain a complete picture of the fault and its effects: the tester would benefit from the behavioral information, returned by RADAR, and the structural information, returned by the fault localization technique. We expect the combination of the two techniques can dramatically improve debugging.

RADAR has not been always successful. In fact, for the open source cases *make* and *indent* the monitoring phase produced slowing downs that prevented practical applicability of the technique. This has been due to the generation of an excessive number of monitoring targets, around 700. Our implementation can reasonably handle up to 500 monitoring points. Thus if the analyzed change is pervasive and targets many long functions, RADAR could generate too many program points. Increasing the size of changes that can be analyzed with RADAR is part of future work, even if according to our experience the case of changes that cannot be analyzed is not frequent (2 out of 10 cases), and the generation of large and pervasive changes is a discouraged software engineering practice.

Threats to Validity

The results that we obtained might not generalize beyond the systems that we analyzed. However, the positive results with both open source and industrial applications suggest that the technique can well address upgrades.

In the empirical evaluation we classified anomalies between true and false positives. The evaluation has been

made by ABB developers for the industrial cases and by academics for the open source cases (both are skilled C/C++ software developers). This classification is subjective and different people could generate different classifications. To avoid to bias the results in favor of RADAR, we classified as true positives only the erroneous events and the erroneous states clearly originated by the fault, and in the ambiguous cases we classified the anomaly as a false positive. In this way the quantitative results should under-approximate the effectiveness of the technique.

The fact that RADAR can detect multiple anomalous events caused by a fault does not necessarily mean that the discovered events are useful to testers. In the empirical evaluation, we did not go into the issue of evaluating if the information extracted by RADAR is enough to fully debug a problem or not. Answering to this question would require a design of a study with human subjects, which we reserve for future work. We believe that the results that we obtained in terms of density of true positives and small distance of the first true positive from the fault location is enough to give the intuition that the chain of events discovered by RADAR is a useful way to backtrack a failure to its causes.

VIII. Related Work

There are many techniques that can be used to analyze a failed execution with the objective of identifying the fault location and the failure causes. Here we discuss the synergies and the complementarities of RADAR with the approaches that work in similar settings. In particular, we consider the approaches that do not require a specification to be applied.

We cluster the related approaches into two categories: fault localization techniques and anomaly detection techniques. Fault localization techniques analyze failed executions to identify fault locations, whereas anomaly detection techniques analyze failed executions to identify the anomalous events that can be responsible for the failure.

Fault Localization: Fault localization techniques analyze the code elements executed by the passing and failing executions to identify the code blocks that most likely include a fault. A well known solution is delta debugging [18], which is an algorithm that can automatically identify a small set of circumstances (e.g., inputs and program statements) that caused a failure. In a recent work delta debugging has been specifically applied to analyze regression problems, demonstrating a good capability of isolating the changes that caused the problem [16].

Statistical fault localization techniques localize faults assuming that the code elements frequently executed by failing executions and seldom executed by passing executions likely include a fault. The many techniques sharing this approach mostly differ for the statistic used to compute the probability that a code block includes a fault [10], [11], [12], [19], [20].

These techniques are useful to localize faults, but do not provide information useful to understand the causes of a failure. RADAR complements to these techniques. In fact, it provides information useful to understand the chain of events that originated the failure. The integration of information about the events that originated the failure and information about the likely fault location provides the best support to debugging tasks, because testers will be able to recognize and confirm the presence of a fault in a location thank to the information about failure causes.

Anomaly Detection: Anomaly detection techniques can be used to analyze a set of executions to identify the anomalous events that occur in these executions. The rationale is that rarely occurring events are suspicious events (likely faulty events) that deserve the attention of testers. For instance, Raz et al. use data models to automatically identify erroneous values returned by online services [21]; Wasylkowski and Zeller use finite state automata to detect improper uses of objects' API [7]; and Hangal and Lam use invariants to identify erroneous variable values [6].

Anomaly detection has been used also to analyze the events occurring in a failed execution. In this case, a set of correct executions are used to distil models that capture the general behavior of a program, then the events occurring in the failed execution are compared with these models to identify the anomalous events responsible for the failure. For instance, BCT is a technique for the analysis of failures in Java programs [9]; KLFA is a technique for the analysis of log files [22]; and AVA is a technique for creating interpretations of anomalous events in a form readable to testers [8].

RADAR is an anomaly detection technique. RADAR differs from the other approaches in the type of faults that it targets. In fact, anomaly detection techniques collect events with a coarse granularity from the entire application. Since the program points that must be used for monitoring are identified a-priori, the effectiveness of the results are largely influenced by this choice. On the contrary, RADAR specifically targets regression problems. In this setting, RADAR has the unique capability of monitoring with a high precision the code related to the change that caused a failure. By exploiting its monitoring layer, RADAR can build precise chains of events that correlate the fault with the erroneous states traversed by the application, until the failure is observed.

IX. Conclusions

Maintaining a software system is an expensive, complex and ever running activity. Among the many activities executed to prevent an upgrade to negatively impact the quality of a system, regression testing is the most commonly used technique [1], [2]. When a test reveals a failure, developers have to analyze the execution to understand the causes of the failure to fix the associated fault.

We have presented RADAR, a dynamic analysis technique, which assists developers in the identification of the erroneous events leading to a failure. RADAR is specifically designed to address regression issues. RADAR focuses the analysis on the behavior of the code regions that are likely affected by the change. When RADAR analyzes a failing execution, it identifies a chain of suspicious events that testers can follow to backtrack the failure to the corresponding fault.

Our empirical results with both open source and industrial systems show that RADAR can effectively assist developers. The feedback from developers using RADAR has shown that the approach not only helps finding faults in the programs, but also understanding the program behavior when the control flow is not explicit (e.g., initialization of static variables or concurrent behavior). In the future, we aim at extending the empirical study and systematically investigate the impact of the size and type of changes, as well as the presence of multiple faults, on the quality of the results.

ACKNOWLEDGMENT

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

REFERENCES

[1] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, 1996.

[2] J. Bible, G. Rothermel, and D. S. Rosenblum, "A comparative study of coarse- and fine-grained safe regression test-selection techniques," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2, 2001.

[3] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2006.

[4] L. Mariani, S. Papagiannakis, and M. Pezzè, "Compatibility and regression testing of COTS-component-based software," in *proceedings of the International Conference on Software Engineering*, 2007.

[5] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the International Conference on Software Engineering*, 2001.

[6] S. Hangal and M. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the International Conference on Software Engineering*, 2002.

[7] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the European Software Engineering Conference and Foundations of Software Engineering*, 2007.

[8] A. Babenko, L. Mariani, and F. Pastore, "AVA: Automated interpretation of dynamically detected anomalies," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2009.

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

[10] J. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the International Conference on Software Engineering*, 2002.

[11] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the Conference on Programming Language Design and Implementation*, 2005.

[12] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, ""SOBER: statistical model-based bug localization"," in *Proceedings of European Software Engineering Conference and Foundations on Software Engineering*, 2005.

[13] S. McCamant and M. D. Ernst, "Predicting problems caused by component upgrades," in *Proceedings of the European Software Engineering Conference and Foundations of Software Engineering*, 2003.

[14] "The DWARF Debugging Format Standard," http://www.dwarfstd.org/.

[15] 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, pp. 99–123, 2001.

[16] K. Yu, M. Lin, J. Chen, and X. Zhang, "Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from developers' perspectives," *Journal of Systems and Software*, to appear.

[17] S. Richter, M. Wahler, and A. Kumar, "A framework for component-based real-time control applications," in *proceedings of the Real-Time Linux Workshop*, 2011.

[18] A. Zeller, "Yesterday, my program worked, today, it does not. why?" in *Proceedings of the European Software Engineering Conference and Foundations of Software Engineering*, 1999.

[19] L. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *Proceedings of the International Symposium on Software Reliability Engineering*, 2007.

[20] M. Renieris and S. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the International Conference on Automated Software Engineering*, 2003.

[21] O. Raz, P. Koopman, and M. Shaw, "Semantic anomaly detection in online data sources," in *Proceedings of the International Conference on Software Engineering*, 2002.

[22] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *Proceedings of the International Symposium on Software Reliability Engineering*, 2008.