Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2011, Article ID 492757, 18 pages
doi:10.1155/2011/492757

Research Article

Bug Localization in Test-Driven Development

Massimo Ficco,[1,2] Roberto Pietrantuono,[3] and Stefano Russo[2,3]

[1] *Dipartimento di Ingegneria dell'Informazione, Seconda Università di Napoli Via Roma, 81031 Aversa (CE), Italy*
[2] *Laboratorio ITeM "C. Savy", Consorzio CINI, Via Cinthia-Edificio 1, 80126 Napoli, Italy*
[3] *Dipartimento di Informatica e Sistemistica, Università di Napoli Federico II, Via Claudio 21, 80125 Napoli, Italy*

Correspondence should be addressed to Roberto Pietrantuono, roberto.pietrantuono@unina.it

Received 30 December 2010; Accepted 28 March 2011

Academic Editor: Hossein Saiedian

Copyright © 2011 Massimo Ficco et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Software development teams that use agile methodologies are increasingly adopting the test-driven development practice (TDD). TDD allows to produce software by iterative and incremental work cycle, and with a strict control over the process, favouring an early detection of bugs. However, when applied to large and complex systems, TDD benefits are not so obvious; manually locating and fixing bugs introduced during the iterative development steps is a nontrivial task. In such systems, the propagation chains following the bugs activation can be unacceptably long and intricate, and the size of the code to be analyzed is often too large. In this paper, a bug localization technique specifically tailored to TDD is presented. The technique is embedded in the TDD cycle, and it aims to improve developers' ability to locate bugs as soon as possible. It is implemented in a tool and experimentally evaluated on newly developed Java programs.

1. Introduction

Test driven development (TDD) is a technique to incrementally develop software, that was sporadically used for decades, and that re-emerged in the last years as development paradigm in the context of the so-called *agile methodologies*.

The increasing adoption of extreme programming (XP) in various industrial projects as well as the identification of TDD as a key strategy in agile software development have captured in recent years an increasing attention also by academic research. As stated by Janzen and Saiedian [1], the TDD is an example of how the academic research sometimes follows the most spread and accepted software practice, rather than leading it. The XP methodology is conceived to reduce the time-to-market of software systems, and the use of TDD practice in XP allows developing more robust programs. Several studies showed this trend, claiming the attention of various researchers, which started to study the ability of TDD to detect software faults earlier in the development process [2–5].

In TDD, the developer writes unit tests from a set of user requirements/functionalities, before writing the code itself. Then s/he implements the code needed to pass the tests, until he succeeds. When a bug is detected, it is promptly fixed. Once the tests are passed, the developer performs the refactoring of the code to acceptable standards; then s/he proceeds to define a new set of test cases for other functionalities and implements a new piece of code to pass them.

In this cycle, even when the tests are successful, the code for new functionalities can easily compromise the previously implemented ones, introducing what are called *regression bugs*. This is not an unusual event in the TDD, since whenever the developer writes the code for some functionalities, s/he does not have a complete view of the whole system in detailed design documents (differently from traditional development cycles) and thus can affect the behavior of the code already implemented.

In order to identify potential regression bugs, the developer, once implemented a group of functionalities, executes regression tests [6], before proceeding to implement new functionalities. To obtain robust programs in low development times, it is essential for developers to be able to correct these bugs promptly.

The described process is particularly suited for relatively small systems. However, when the size and complexity of the system significantly increase, locating regression bugs manually becomes a very challenging task. When regression tests reveal a failure, the debugger should track the bug

propagation chain by repeatedly running the tests step by step, until s/he identifies the bug source. If the system is large and complex, with many interacting units and intricate interdependencies, the effort to locate a bug can become unacceptable. In this phase, a tool to improve the debugging process would allow the TDD practice to yield high productivity also in large projects.

In this paper, we propose a technique to speed up the bug localization activity in the TDD process. It is based on an anomaly detection approach that is iteratively applied to each TDD cycle. The basic idea is to compare at each cycle the behaviour of the application observed during regression testing with a reference "correct" behavioral model as intended by the developer, that is, a model describing how the application should behave. In particular, when a failure occurs, the technique identifies any deviation from the reference model and then highlights the ones most likely related to the bug.

For this aim, we implemented a tool that collects deviations from the expected behavior and assigns them a weight measuring their likelihood of representing the bug. Evaluation metrics are defined and implemented in order to assign such weights and rate the identified anomalies. Based on this ranking, the debugger inspects directly those points that are most likely related to the bug.

The implemented metrics are evaluated by applying the tool to a known open-source Java library, namely *JFreeChart* (JFreeChart (version 1.0.9) is a free Java chart library to develop professional quality charts. It is available at: http://www.jfree.org/jfreechart/), using a fault injection technique to reproduce the presence of bugs in the code, and considering the goodness of the final deviations ranking.

The overall approach and the implemented tool are also evaluated by running a set of experiments in which several groups of students developed a new application from scratch (from the same set of requirements) by using the TDD approach. In these experiments, the debugging times, measured during the development without the support of the proposed technique, are compared with the tool-supported debugging times, in which developers employed the tool at each TDD cycle. Results show a significant improvement of debugging times across TDD cycles.

The rest of this paper is organized as follows: Section 2 outlines the background about the TDD practice; Section 3 describes the proposed solution and provides some implementation details. Section 4 shows the experimental evaluation and discusses the obtained results. Finally, Section 5 is devoted to related work, and Section concludes the paper.

2. TDD

TDD consists of few basic steps iteratively repeated over time, with the aim of translating user requirements into test cases, that guide the developer in the code implementation process. It does not replace traditional testing, instead it defines a proven way to ensure effective unit testing and provides a working specification for the code. Figure 1 shows the main steps of a TDD process. They may be summarized as follows.

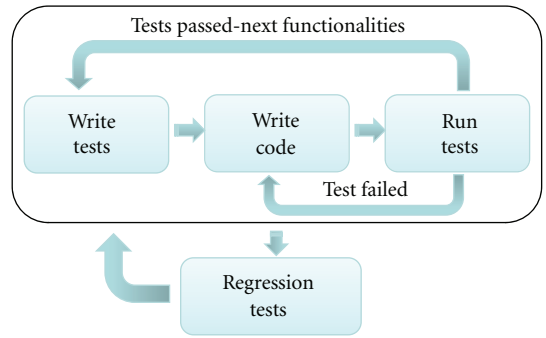


Figure 1: TDD process.

(1) The developer writes an automated test case that defines a new functionality to be implemented.

(2) Then he writes the code necessary to pass the defined test.

(3) The developer runs the test. If the test gives negative outcome, he refactors the new code and skips to the next functionality. Otherwise, he refines its code until the test is passed.

(4) The developer periodically reruns all the written test cases, to be sure that the new code does not cause failures in the previous code (the previously implemented functionality). This phase is called *regression tests*.

In the literature, the features of TDD have been shown to bring benefits not only by reducing the development time, but also by producing more robust code [5]. Several empirical studies have been conducted on the effectiveness of TDD practice, both by industry and by academia. Williams et al. [7] have shown how TDD produced, in their experiments, higher-quality code. In the same year, George and Williams [8] reported reduction in defects density up to 40% compared to an ad hoc testing approach, while in [6] and [3], the authors reported, respectively, quality improvements of 18% and 50%, with an impact on the development time of 15–20% (i.e., they showed that sometimes TDD adoption took longer) with respect to a traditional development. Another study [9] confirmed these results, reporting an even higher reduction in faults density. However, results in [4] show that TDD helps in reducing overall development effort and in improving developer's productivity, but the authors observed that the code quality seems to be affected by the actual testing efforts applied during a development style. There are other studies, like the one conducted by Geras et al. [10], or two other academic ones [11, 12], that observed no significant differences in the developer's productivity, even if they involved programmers who had little or no previous experience with TDD. The authors highlighted the need to improve testing skills of the developers for effectively adopting test-first approaches.

2.1. Regression Tests. An important role in the TDD is played by regression tests. Unlike traditional development cycle, in

the TDD, regression bugs are very likely to be introduced during the work cycles, since when a developer implements a functionality, s/he does not have a complete view of the system to develop in its detailed design (e.g., due to the lack of a design documentation, or because the user requirements are not yet fully defined). The developer does not clearly know how the successive functionalities will affect the current one. Hence, changes in a code area may inadvertently cause a software bug in another area, and a function that worked correctly in a previous step may stop working.

The detection of regression bugs is therefore a crucial step in TDD, which occurs "inside" the development, not after. For instance, let us suppose that we are developing a simple software application (e.g., an online bookshop) made up of 7 functionalities, and that we have already developed the first two functionalities (see Algorithm 1).

For simplicity, suppose that the functionality F2 is responsible for shipping an order; it will include a method *ship (ISBN)* that needs the current number of available items with the given ISBN, denoted with *currentN*. The management of the current availability of books is devoted to another functionality, F6, not yet implemented. F6 will be in charge of verifying the current availability (by the method *verifyAvail*) and of updating it (the method *updateAvail*).

During the implementation of F2, test cases give negative outcome; hence, the developer skips to develop the third functionality. F2 test cases assume that the *currentN* value comes from F6 methods (i.e., they use a *driver*, since F6 is still not implemented). Based on the number of ordered items and on the *currentN* value, it ships the order (and updates the availability, temporarily through a *stub*) or returns an error.

The developer then proceeds in the development up to the sixth functionality, responsible for managing the items availability. During the F6 implementation, the developer realizes that the *updateAvail* method needs to distinguish between a *sale* order, by a client, or a *purchase* order by the bookshop manager to increase the current availability. Then s/he introduces a boolean variable (*SalePurchase*) to discriminate the two cases. As a consequence of this unforeseen requirement, s/he has to modify F2 code. The modified code is reported in part 3 of Algorithm 1 . As may be noted, the developer makes a mistake during the F2 modification, introducing a bug: a wrong logical condition in the first *if* statement is specified (an AND operator instead of an OR). Test cases for F6 give negative outcom; hence, the developer skips to the seventh functionality. The problem comes up only when the developer executes regression tests. When the developer implemented the *ship* method in F2, s/he did not have a clear view of F6 and did not handle the case causing the failure.

An example of more subtle bugs is in the F6 code; during the F6 implementation, the developer makes a mistake in the computation of the variable *currentN* (used by F2) in the function *updateAvailability*. In particular, s/he omits a check on the maximum capacity of items that can be stored. Test cases for F6 give negative outcome, since they have only to check that the availability is correctly updated. Hence, the developer skips to the next functionality; however, the variable *currentN* can assume a wrong value (greater than the maximum capacity), which can bring to erroneous behaviours of the F2 functionality. Again, the problem comes up only when the developer executes regression tests, when s/he can realize that the variable *currentN* may be in an inconsistent state.

In a complex system, the situation is typically worse than the exemplified ones, since from the bug activation to the failure manifestation, there may be a very long and intricate chain (e.g., the variable *currentN* of the example could have been used by other functionalities without causing failures up to F2); tracking back from the failure to the bug is not trivial. The present work aims to ease the identification of this kind of bugs that are not detected by unit tests, and that typically are the main cause of development time wasting in the TDD cycle.

3. Anomaly Detection Approach

The use of a debug-aiding technique in the TDD process allows guiding the debugger through the identification of potential root causes of failures introduced during the development, as the one reported in Algorithm 1.

In the following, we refer to a failure occurrence as a consequence of a chain *fault-error-failure*, in which a software fault, once activated, becomes an error, propagating through the system up to the interface. As bugs in the code are software faults, just for the purpose of this work, we use the term *software fault* or simply *fault* interchangeably with the term *bug*. During the fault propagation, the system exhibits some behaviors different from the expected ones. We call such deviating behaviors *anomalies* or *violations*. In the described example, the omitted control in the *updateAvail* method represents the bug, which, once activated (i.e., the control flows through the method), propagates up to the *ship* method and becomes a failure when an order is shipped with a wrong *currentN* value (greater than the number of books actually available). In that case, a *violation* may be a value of the *currentN* variable at the exit point of the *getCurrentN* method different from expected (e.g., outside an expected range).

From the bug activation to the failure manifestation, there is, in general, a chain of one or more violations. Highlighting such violations may be very important for debugging purposes, since they point out where the bug is propagating. However, identifying a violation requires (i) a clear description of the system expected behavior and (ii) the definition of what a deviation from this behavior means. Moreover, since there can be a high number of violations and very complex violation chains, we need a way to distinguish the most relevant violations (i.e., the most related to the failure-causing bug). Thus, a tool should support the tasks of (i) building an accurate model of the expected behavior of the system, (ii) identifying violations with respect to the built model, and (iii) discriminating the most relevant violations (according to their proximity to the bug). This support would lead the debugger to inspect directly the points closest to the bug, thus reducing debugging (and development) time.

(1) Functionality F2::shipOrder

```
...
orderData getOrderData() {...}
void ship(sellOrderData) {
if(orderData.Number < 0 || !(checkISBN(orderData.ISBN))
  {// return error}
N = F6::getCurrentN(orderData.ISBN);
if((N − orderData.Number) < 0) {
    //return error
    ...
}
else {
    //ship order
    ...
    F6::updateAvail(sellOrderData)
    ...
}
}
//other methods to support the shipment
...
```

(2) Functionality F6::ManageAvailability

```
Boolean verifyAvail() {...}
updateAvail (orderData, SalePurchase) {
if (SalePurchase == ''Sale'') {
  if ((currentN − orderData.Number) < 0) {
     //return error
     ...
  }
else {
    //execute update
    currentN = currentN - orderData.Number
    ...
  }
}
else if (SalePurchase == ''Purchase'') {
     currentN = currentN + orderData.Number
//!! ---check on Maximum Capacity missed---- !!
    ...
  }
}
else {... //error}
...
}
// other methods
...
```

(3) Functionality F2::shipOrder, modified after F6 Implementation

```
string SalePurchase = ''sale'';
orderData getOrderData() {...}
void ship(sellOrderData) {
if(orderData.Number < 0 || !(checkISBN(orderData.ISBN))
        && !(stringIsValid(SalePurchase))
  // return error
//!---wrong logical condition ----!
...
N = F6::getCurrentN(orderData.ISBN);
if((N - orderData.Number) < 0) { //return error}
else { //ship order
     F6::updateAvail(sellOrderData, SalePurchase) }
...
} //other methods to support the shipment
```

Algorithm 1: Regression bugs in an online bookstore.

In this work, the description of the expected system behavior relies on a dynamic analysis technique that derives behavioral models from execution traces. The behavioral model is described by a set of execution traces reporting the interactions among methods. The interactions are described in terms of *invariants* and *call sequences*. An invariant is a relation describing a property that should be always preserved. In particular, we consider input/output (I/O) invariants, which can be inferred from tests execution by observing the exchanged argument values. For instance, an invariant involving a single variable x may require it to be comprised in the range $[a, b]$; an invariant involving two variables may require they to respect the relation $x < y$. A behavioral model is therefore made up of a set of I/O invariants and a sequence of function calls (reported in a trace file). For building such models, we consider the technique presented in [13], which produces a trace file representing the behavioral model that reports the sequence of calls and invariants on I/O method's arguments, built (and updated) at each entry/exit point of methods. Note that the iterative nature of TDD process favors the adoption of techniques based on invariants construction, since it allows building more and more accurate behavioral models as development proceeds across TDD cycles. *A **violation** to this*

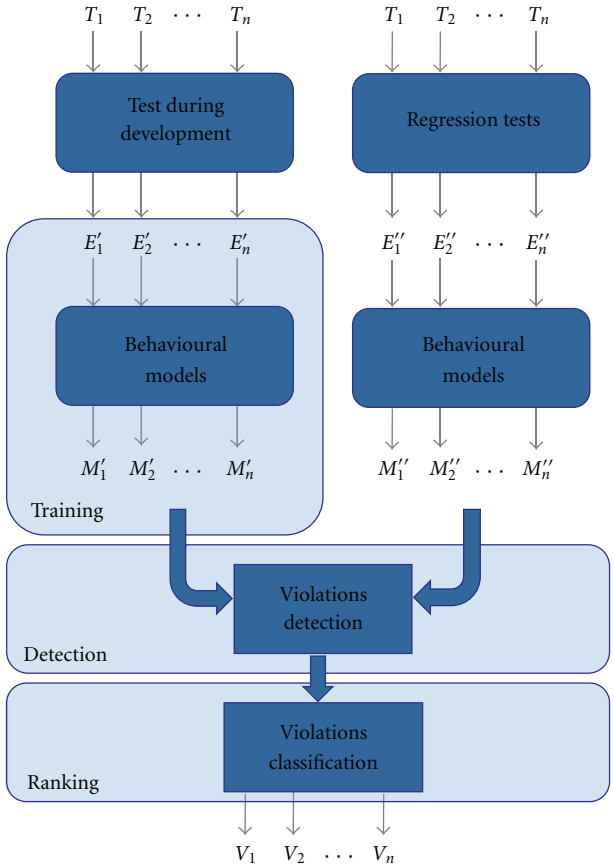


FIGURE 2: The proposed approach. T_i are test cases, E_i are execution traces resulting from tests execution, and M_i are the inferred models. V_i are the final list of violations.

model is caused either by a deviation from the built invariants (i.e., an argument value that violates the property described by the invariant) and/or by a sequence of calls different from the expected ones. For instance, for an I/O invariant over a variable x such that $a < x < b$, a violation is caused by a value of x outside the range $]a, b[$.

As shown in Figure 2, the solution we propose is based on an anomaly detection approach that acts in three phases: *training*, *detection*, and *ranking*, which are applied to each TDD cycle. In the *training* phase, the behavioral model of the application representing the expected "correct" behavior is built. In the *detection* phase, an anomaly detection method is adopted to highlight the deviating behaviors (i.e., *violations*) with respect to the correct behavior. Then, in the *ranking* phase, all inferred violations are rated, in order to identify the violations closest to the bug.

Training. TDD steps include writing tests for a given group of functionalities, and then writing code to pass the them. Tests are executed until success, before skipping to the next functionalities. Training takes place during the (successful) execution of these tests, that represent the desired behavior for those functionalities. In particular, while executing these tests $(T_1, \ldots, T_n)$, traces $(E'_1, \ldots, E'_n)$ are gathered and a series of behavioral models $(M'_1, \ldots, M'_n)$ are built. The models, obtained from the test executions (only from passed tests), represent the behaviors that are expected for the functionality being tested, that is, the desired and hence "correct" behavior.

Detection. After that, a group of functionalities is implemented; in the last step of a TDD cycle, regression tests are executed (with the already available test suites). During this step, execution traces are gathered again and compared to the built models to detect violations. In practice, the call sequences and the argument values are observed and compared with the call sequences and the I/O invariants of the correct behavioral models. Thus, if some code fragment, written to implement a new functionality, has altered the behavior of a previously implemented one, or if a bug introduced in a functionality (and not correctly fixed) causes a behavior different from the expected one, the technique will capture these *violations*.

Ranking. During the detection phase, different violations are inferred. We call "*bug-related*" violations all those violations directly related to a bug (i.e., that are consequence of bug activation). The remaining violations that are not related to a bug represent *false positives*. In order to discriminate false positives, as well as to reduce the time required to analyze the volume of bug-related violations, we propose a weight-based solution for ranking. In particular, the detected violations are rated according to their likelihood of being the closest violation to the bug that caused the failure.

3.1. Ranking. As previously stated, the adopted anomaly detection approach does not directly detect the bug, but it identifies violations that appear along the fault-failure chain. However, such an approach may generate a large volume of violations, and no diagnosis is provided in order to help the debugger to identify whether a violation is a true positive (i.e., bug related) or a false positive (i.e, behaviours not related to the bug). Moreover, in the case of true positives, no information is provided to determine which violation is closest to the bug activation. The goal of the ranking phase is to rate *bug-related* violations according to their closeness to the bug activation. In this process, more distant bug-related violations have to be rated in the lower part of the ranking; *fals positives*, which are not related to the bug, should be rated even lower, in the last positions. For this aim, we defined a metric that assigns a value to violations representing their likelihood of being the closest one to the bug.

In particular, we consider three criteria to determine the final probability values:

(1) we consider that during the execution of regression tests, if a bug is present, it may cause more than one test case execution to fail. Hence, we will have a set of execution traces corresponding to the failed tests. In this case, those violations that are common to more executions with failing outcome are deemed more likely to be related to the failure-causing bug. Essentially, if a violation (e.g., an argument value out

of the expected range) is always present when the system fails, then it will be more likely related to the bug;

(2) given a violation v_i, we consider, as additional criteria to determine the final probability value of v_i, the number of violations present in the failed execution traces containing v_i;

(3) the occurrence position of v_i in such traces.

Trying to translate these qualitative considerations into a metric, we define, given a violation v_i, the following parameters:

(1) the number of failed executions in which v_i is present (Nf) over the number of executions in which v_i is present (N). The violation v_i is as much likely to be a *bug-related* violation as the ratio Nf/N is close to 1. In the case that the ratio amounts to 1, this means that each execution containing v_i failed;

(2) for each failed execution, we consider the ratio $1/nP$, where nP is the number of violations in that execution. If v_i was the only violation in that execution, it would be very likely related to the bug causing the failure. The less the number of violations, the more likely a violation is deemed responsible for (i.e., directly related to) the failure;

(3) for each failed execution, we consider the position of v_i in the execution trace with respect to the other detected violations. Also in this case, we consider the ratio $1/\text{pos}$, where pos is the position (starting from 1 to n, with n being the number of violations in the considered execution). If v_i is the first violation in the considered failed execution, it will be not a consequence of other violations (thus, it is considered more likely related to the bug). On the other hand, when the position of v_i is closer to n, it is more likely a consequence of other violations (thus, it is less likely the closest one to the bug activation).

Taking the extreme case, v_i has probability 1 to be related to the bug if the following conditions contemporarily stand:

(1) $Nf = N$, that is, each time v_i is present in an execution, that execution fails;

(2)

$$\sum_{\text{executions}} \left[pF * \left(\frac{1}{nP} * \frac{1}{\text{pos}} \right) \right] = Nf \text{ with}$$
$$pF = 1 \quad \text{if the execution containing } v_i \text{ failed,} \tag{1}$$
$$pF = 0 \quad \text{otherwise.}$$

Equation (1) means that for each failed execution ($pF = 1$), which contains v_i, v_i is the only present violation, which implies it in the first position (i.e., $nP = 1$ and pos = 1).

In these conditions, the contributions 1 and 2 will be both equal to N. Therefore, the metric can be formalized, with respect to the extreme case, as follows:

$$p_i = P(v_i \text{ is bug-related}) = \frac{Nf + \sum_{\text{executions}} \left[pF * \left((1/nP)^{\alpha} * (1/\text{pos})^{\beta} \right) \right]}{2N}, \tag{2}$$

that is 1 in the described extreme conditions. The factors α and β (both comprised between 0 and 1) are needed to weigh the importance of the parameters 2 and 3, respectively (i.e., the number of violations and the relative position).

Referring to the example presented in Section 2.1 (Algorithm 1), suppose that the bug introduced in the F6 functionality produces a value of the *currentN* variable, that violates an I/O invariant built on the method *getCurrentN* of F2 (i.e., a value out of the range observed during the model construction phase). Assume that this violation appears in three different executions during the regression tests, and that two of these executions fail (i.e., $Nf = 2$ and $N = 3$). Now, assuming that the violation appears in the two failed executions in the positions pos = 10 and pos = 5, respectively, and that for these executions $nP = 120$ and $nP = 50$. In such a case, the probability value assigned to the violation wo-uld be

$$P(v_i \text{ is bug-related}) = \frac{2 + \left[(1/120)^{0.3} * (1/10)^{0.5} \right] + \left[(1/50)^{0.3} * (1/5)^{0.5} \right]}{6} = 0.369. \tag{3}$$

In this example, the main contribution is given by the number of failed executions in which v_i is present (two out of three) that is 0.333. Instead, the high number of violations in both failed executions causes a very little contribution (0.035). At the extreme, if v_i was the only present violation in the two failed executions, the value would be 0.666, which is the maximum value assignable with two failed executions over three. The conditions (1) and (2), that correspond, respectively, to the first and the second terms of (2) numerator, both contribute to the final probability with a value between 0 and 0.5. In this case, with two failed executions containing v_i, their contribution is at most 0.333 for both. Thus, the value Nf, other than determining the first contribution, also limits the second contribution, since the latter considers only failed executions. An improvement to this metric can be brought by considering that it does not take into account potential relationship among violations referring to the same method. More violations referring to the same method could indicate a high likelihood that the fault is located in that method. Thus, a further refinement is possible; all the violations referring to the same method (even with different I/O violations on the exchanged

parameters) are grouped together, and for each method M in the execution trace, the following value is assigned:

$$M\text{-value} = \frac{\sum_i p_i}{\sum_j p_j}, \quad (4)$$

where p_i is the probability value assigned to the violation v_i. The index i belongs to the set of all violations referring to the method M, and j refers to all the violations. In this way, a ranking with respect to the likelihood for the method to exhibit the violation closest to the bug is done, rather than on the single closest violations. This second version of the metric has shown significant improvements, as described in the experimental section.

With this schema, other refinements are possible. For instance, in order to further penalize those violations with lower probability, a third version is obtained by considering each probability p_i divided by the relative position of v_i (considering just the ranking relative to the method M). For instance, consider two methods A and B. Suppose that method A is responsible for the bug, and the second version of the metric reported one violation rated in the first position of method A's ranking (e.g., v_A^1 with probability $p_1 = 0.9$), and method B has two violations (e.g., v_B^1 and v_B^2 with probabilities $p_1 = 0.7$ and $p_2 = 0.3$, resp.), in the first and second position of method B's ranking. The *M_value* numerator assigned to A with the second metric would be 0.9, and the *M_value* numerator for B would be $0.7 + 0.3 = 1$. Instead, by considering the relative position (third version of the metric), the numerator for A will still be 0.9, but the numerator value assigned to B will be $0.7 + 0.3/2 = 0.85$, penalizing the second violation in B; this would correctly identify method A as the most probable related to the bug.

More drastic refinements could be considered, by further penalizing the violations that are in the lowest part of the ranking, but they would not always yield the right result. At the extreme, the risk is to cancel the positive effect introduced by the second version. Thus, in the experimentation, we took into account only the three described versions of the implemented metric, referring to them as first, second, and third metrics.

3.2. Implementation. A prototype tool embedded in the TDD cycle has been implemented to evaluate the performance of the described technique. Figure 3 represents the tool high level architecture. We named the tool ***ReTest***.

The first block is the *model builder*. It is responsible for instrumenting the application, monitoring it, and building, from the execution traces, the behavioral models. For each execution trace, it builds and stores a model. In particular, the *Model builder* block uses the tool *Daikon* [13]. *Daikon* is a dynamic analysis tool that derives invariants from execution traces. It infers a set of invariants by observing the exchanged parameter values and by statistically comparing them with a set of 160 predefined templates. It starts with a set of syntactic constraints for the considered variables and incrementally considers the input values. At each step, it eliminates the constraints violated by the value to obtain a set of constraints satisfied by all inputs. Statistical considerations allow Daikon

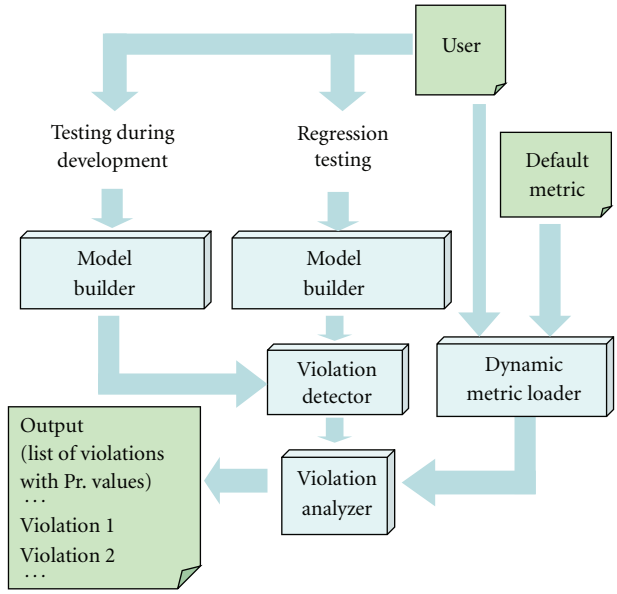


Figure 3: The tool architecture.

to identify constraints that are verified incidentally. The output of *Daikon* is a list of I/O invariants for each of the instrumented points.

The behavioral model is therefore described by a set of execution traces reporting the sequence of calls and invariants built (and updated) at each entry/exit point of methods, and stored in a file. Figure 4 shows a simple example developed by Daikon's author [14] describing invariants built on a Java class `StackAr` that implements a stack with a fixed maximum size. The class has two fields, namely, `Object[]theArray` and `int topOfStack`, representing, respectively, the array that contains the stack elements and the index of the top element (with -1 meaning stack empty). Among the class methods, Figure 4 shows invariants produced for the `isFull` method.

The output for each invariant is described below:
`this.theArray != null`: the reference `theArray` was never observed to be null after it was set in the constructor.
`this.theArray.getClass() == java.lang.Object[].class`: the runtime class of `theArray` is `Object[]`.
`this.topOfStack >= -1`, `this.topOfStack <= this.theArray.length - 1`: `topOfStack` is between -1 and the maximum array index, inclusively.
`this.theArray[0..this.topOfStack] elements != null`:
all of the stack elements are nonnull, that is, the test suite never pushed null on the stack.
`this.theArray[this.topOfStack+1..]elements == null`:
all of the elements in the array that are not currently in the stack are null.
Likely invariants are also found at each method entry and exit. These correspond to the pre- and postconditions for

the method. There is one precondition for the StackAr constructor: capacity >= 0: StackAr was never created with a negative capacity.
The postconditions for the StackAr constructor are
orig(capacity) == this.theArray.length: the size of the array that will contain the stack is equal to the specified capacity.
this.topOfStack == −1,
this.theArray[] elements == null: initially, the stack is empty, and all of its elements are null. There are no preconditions for the isFull method other than the object invariants. The postconditions are as follows
this.topOfStack == orig(this.topOfStack),
this.theArray == orig(this.theArray),
this.theArray[] == orig(this.theArray[]):
neither the topOfStack index, nor the reference to the theArray array, nor the contents of the array are modified by the method.
(return == true) <==>
(this.topOfStack == this.theArray.length − 1):
when isFull returns true, topOfStack indexes the last element of theArray.
(return == false) <==>
(this.topOfStack < this.theArray.length − 1):
when isFull returns false, topOfStack indexes an element prior to the last element of theArray.

Models are traces containing such kind of invariants in sequence for each instrumented method call and that are iteratively updated.

The second block is the *violation detector*, which is responsible for detecting differences between the models obtained in the training phase and the models obtained in the detection phase. The *violation detector* block is based on a tool related to the *Daikon* suite, that is *InvariantDiff*, whose goal is to detect violations to a set of I/O invariants. This block detects both I/O invariant violations, supported by *InvariantDiff*, and violations to the sequence of calls that have been recorded.

Violations are stored and then analyzed by the *Violation Analyzer* block. This block rates violations to determine their likelihood of being related to a bug. The block implementing the three described metrics is the *Default Metric* block. Finally, the tool design also includes a *Metric Loader* block to allow the user to define and adopt its own metric. The output is a list of violations reporting the involved methods and variables and the corresponding probability values assigned by the metric.

Figure 5 shows a screenshot of the tool, during experiments, displaying the output file that reports the final list of violations. Information reported in this output file is the name of the violation detected, the name of the involved package, class and method, and the values of each parameter used by the metric (i.e., Nf, N, the positions of violation in the failed execution traces, the number of violations in failed executions, the values for α and β, and the final probability value assigned to that violation). In the example, the reported violation had a probability of 0.351 to be the closest one to the bug activation (i.e., it is the one most likely related to the bug). Debugger would start its debugging process from the first violation reported in this output file.

The implemented tool uses Java as development language, Eclipse as development environment, and JUnit to run test cases. Hence, it can be executed during development in conjunction with JUnit framework and seamlessly added as a plugin to the Eclipse IDE. This aspect favors the usability of the tool, since both Java and JUnit are widely used and well known in both industry and academia. In our experiments, students confirm these considerations; however, the tool is a prototype and has been used only by students; hence, any systematic evaluation of usability would still be premature.

As for portability, being based on JUnit and on a *Daikon* Java front end (i.e., *Chicory*), the implemented tool currently works only with Java programs. However, future extension is straightforward, since it is sufficient to use the *xUnit* family tools and the other Daikon front ends to support other languages too (such as C/C++).

4. Experimentation

The proposed ranking metrics are evaluated by emulating both the development of additional functionalities to an existing application (i.e., an extension to an existing application) and the presence of various types of faults (by a faults injection campaign). These experiments allowed us to evaluate the three metrics by considering the goodness of the ranking, that is, the closeness of the first rated *bug-related* violation to the actual bug. A comparison with a well-known algorithm for debugging, namely SOBER [15], is also provided. Once evaluated the ranking ability of the proposed metrics, the technique is also experimented on an application developed from scratch by using the TDD approach.

4.1. Metric Evaluation. To evaluate metrics performance, *ReTest* is applied to a well-known open-source Java library, namely *JFreeChart* (JFreeChart (version 1.0.9) is a free Java chart library to develop professional quality charts. It is available at: http://www.jfree.org/jfreechart/), being available both the source code and the unit test cases. JFreeChart source code consists, in the chosen version, of a set of about 560 classes and 7500 functions, amounting to nearly 200 K Lines-of-Code (LoC). *ReTest* is applied to a subset of this code in the packages org.jfree.chart.util and org.jfree.chart.renderer, corresponding to 2618 LoC.

4.1.1. The Procedure. In order to evaluate the effectiveness of the three metrics in a TDD process, we emulated the development by executing the four steps of TDD and by emulating the presence of faults via *fault injection*.

The first two steps of TDD (i.e., "Write Tests" and "Write Code") do not require relevant efforts, since both the tests and the source code are available. The third and fourth steps (i.e., "Run Tests" and "Regression Tests") are carried out by using the *JUint* framework. In the regression test phase, *ReTool* evaluates the violations caused by injected bugs and

Object invariants for StackAr

```
this.theArray != null
this.theArray.getClass() == java.lang.Object[].class
this.topOfStack >= −1
this.topOfStack <= this.theArray.length - 1
this.theArray[0..this.topOfStack] elements != null
this.theArray[this.topOfStack+1..]elements == null
```

Preconditions for the StackAr constructor

```
capacity >= 0
```

Postconditions for the StackAr constructor

```
orig(capacity) == this.theArray.length
this.topOfStack == −1
```

Postconditions for the isFull method

```
this.theArray == orig(this.theArray)
this.theArray [] == orig(this.theArray[])
this.topOfStack == orig(this.topOfStack)
(return == false) <==>
 (this.topOfStack < this.theArray.length − 1)
(return == true) <==>
 (this.topOfStack == this.theArray.length − 1)
```

Figure 4: An excerpt of invariants of the `StackAr` program.

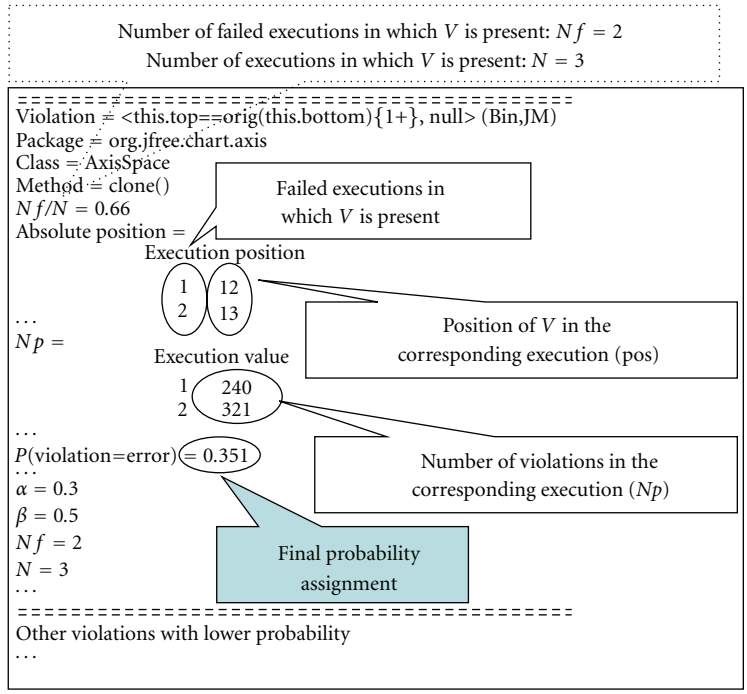


Figure 5: The output file reporting the violations list.

rates them according to the defined criteria. In particular, the experiments are performed as follows:

(i) as a first step, we choose a group of functionalities (and the corresponding unit test cases to be run) that represent the functionalities to be developed;

(ii) then we manually inject a fault in the code, by paying attention to not inject faults detectable by unit test cases (because in this case, the fault is supposed to be removed before completing the functionality implementation, and thus prior to the regression tests). If injected faults are detected by unit test cases, the experiment is repeated by changing the injection;

(iii) once injected the fault, the unit test cases corresponding to the chosen functionalities are run (with the instrumentation of our tool), in order to verify if the code passes the tests. During unit tests execution, the behavioral models are built;

(iv) when unit tests do not detect any faults, development of that functionality is supposed to end. Then, regression tests are performed over all the functionalities developed till then, since the injected bug can cause anomalies in previously developed functionalities, as argued above. During regression tests, the tool automatically collects violations, computes metric values, and classifies them according to their likelihood of being related to the injected bug.

Thus, the output of the last step is a list of violations, which can be either *false positives* or *bug-related* violations. The debugger starts in such a bug report from the violation rated in the first position by the tool. If it is a *bug-related* violation (i.e., a consequence of the bug), s/he locates the bug by inspecting the code starting from that violation. If it is a *false positive*, s/he is not able to locate the bug and, after some time, skips to the second violation, until s/he meets a *bug-related* violation. Hence, it is possible to consider two evaluation measures:

(1) the first one is the position of the first *bug-related* violation in the ranking carried out by the tool. The higher this position, the higher the debugging time;

(2) we consider the *violation chain*, VC $= \{v_1, v_2, \ldots, v_n\}$, as the sequence of violations in the execution trace that occurred from the bug activation to the failure. Let V denote the bug-related violation reported by the tool in the first position of the bug report, as the closest one to the bug according to its ranking. To evaluate how much this ranking is correct and useful, the second measure is the *length* of the violations chain between the bug and V, we call it *VC length*. Since debugger would start from V to localize the bug, the higher *VC length*, the more time the debugger will employ to locate the bug (more code needs to be inspected).

The latter measure turned out to be much more important than the position of the first *bug-related* violation, since in almost all the performed experiments, the tool rated in the first position a *bug-related* violation (i.e., false positives are almost always rated in the lowest positions). Actually, if the tool rates in the first position a false positive, the tool performance has to be considered not acceptable (as showed in the next section), since the debugger would be misled and would spend too much time before skipping to the second violation. When the first violation is a *bug-related* one (in almost all the cases), the relevant evaluation measure is therefore the VC length in the execution trace.

4.1.2. Fault Injection. Fault injection is a common approach to evaluate debugging techniques (e.g., [15–21]). There are several approaches to emulate software faults. Faults can be reproduced either by modifying the source code of the target application, that is, by injecting the actual fault (software mutation, SM), or by injecting errors. Software mutation has been showen to be the most accurate way to emulate software faults [22]. It can be applied when the source code is available, and it allows to emulate all the software faults encompassed by a well-known classification scheme, that is, the orthogonal defect classification (ODC, [23]).

Table 1: Kind of injected faults.

ODC type	Description
Function	Affects significant capability (such as end-user interfaces or global data structure) and requires design change
Assignment	Affects a few lines of code (such as initialization of control blocks or data structure)
Checking	Addresses program logic that has failed to validate data and values
Algorithm	Includes efficiency or correctness problems that affect the task and requires reimplementation

For these reasons, we preferred SM to error injection techniques, such as SWIFI, and to binary mutation also known as G-SWFIT [24]. Indeed, SWIFI does not allow to reproduce software faults that require large modifications to the code, or that are due to design deficiencies, such as *function faults* (see [23]). By G-SWFIT, it is tricky to assure that injection locations actually reflect the high-level constructs where faults are prone to appear. Moreover, it cannot assure that binary mutations are the same as generated by source code mutations. Hence, we inject faults directly into the source code (according to the ODC classification). Table 1 indicates the ODC types that have been injected and their meaning.

4.1.3. Results. We applied the outlined procedure for an experimental set of 20 different faulty versions and evaluated the performances of the three implemented metrics. Experiments produced the results reported in Table 2, for the three implemented metric versions. In particular, Table 2 shows, for each experiment, the *VC length* between the bug activation and V, (i.e., the *bug-related* violation reported in the first position by the tool), and the number of violations detected during each experiment. Note that more violations may refer to the same method called several times. Experiments marked with a "$*$" character are the ones in which the tool rated as first violation a false positive (in which cases the tool performances are considered as not acceptable). In all the other cases, the first rated violation is a *bug-related* violation.

Results show that best performances are obtained by the third metric. In 75% (15 over 20) of the experiments has the *VC length* is less than six. Compared to the first metric, the latter shown the same results for just 6 experiments (30%), while the second metric for 14 over 20 experiments (70%). In the remaining cases (resp., 25%, 70%, and 30%), the *VC lenght* is always greater than ten, thus significantly compromising the ability of the technique to locate bugs in acceptable time (also in these cases, the third metric shows better results).

Table 2: Results showing the *VC length* in the execution trace from the bug activation to the *bug-related* violation V and the number of total violations in each test.

Test	VC length/Tot. violations by metric 1	VC length/Tot. violations by metric 2	VC length/Tot. violations by metric 3
1	73/596	1/596	1/596
2	1/77	2/77	1/77
3	39/293	3/293	3/293
4	142/168	133/168	126/168
5	49/301	1/301	1/301
6	211/291	13/291	13/291
7	109/172	3/172	3/172
8	87/231	5/231	5/231
9	1/46	22/46	22/46
10	1/31	1/31	1/31
11	16/56	4/56	4/56
12	21/133	6/133	6/133
13	23/129	5/129	5/129
14	276/312*	211/312*	19/312*
15	37/93	2/93	2/93
16	1/43	2/43	1/43
17	2/77	1/77	1/77
18	123/149*	61/149*	33/149*
19	16/211	103/211	3/211
20	1/37	1/37	1/37

4.1.4. Considerations. Results show that the introduction of the second criterion (i.e., violations grouped according to the method they refer to) has been successful. A further improvement has been obtained by giving less importance to the violations rated at the lowest part of the ranking (i.e., the third metric), bringing the most of the *VC length* values under six. In the experimentation phase, described in the next section, we adopted the third metric, which provided the best results.

From the analysis of the injected faults, it has been also possible to observe the metrics behaviors with respect to different fault types. The diagram in Figure 6 represents the number of violations whose VC length is in the range 1–6 (i.e., the best results), classified per injected fault type and per used metric.

The three metric versions behave almost in the same way for *assignment* and *checking* fault type. This means that grouping violations by methods is quite irrelevant for these types of fault. The second and third metrics introduce improvements in the *function* and *algorithm* category. The changes introduced by these metrics affect the ability to localize faults related to the "logic" of the code (e.g., "Algorithm"). The most relevant improvement introduced by the third metric is about the *function faults*. This type of faults affects the design, requiring design changes and large modifications to the code. Grouping by methods and giving more importance to the first violations led to better localization of these more complex faults. This can be explained by considering that faults requiring significant design changes very likely cause more violations for the method where they are injected.

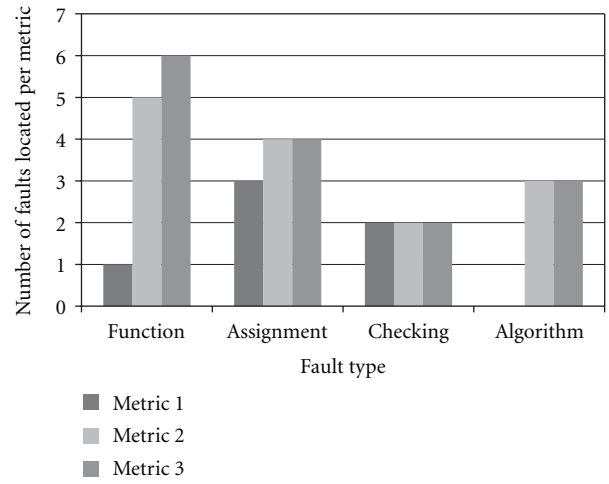


Figure 6: Number of violations (in the range 1–6) per fault type and per metric.

4.1.5. Comparison with SOBER. Performances of the ranking algorithm are compared with a known debugging algorithm, called SOBER [15]. Sober is a statistical approach to localize bugs that models evaluation patterns of predicates in both correct and incorrect runs. It regards a predicate as bug relevant if its evaluation pattern in incorrect runs differs significantly from that in correct ones. In particular, given a set of instrumented predicates, SOBER computes for each predicate P a score indicating its bug relevance, based on values taken by P in both correct and failing executions. Observing the values of P, its *evaluation bias*, representing the probability that P is observed as *true* in each evaluation, is computed as follows: $\pi(P) = n_t/(n_t + n_f)$, where n_t is the number of times P is evaluated to true in one execution, and n_f is the number of times P is evaluated to false.

Considering multiple executions, SOBER computes the probability density function (*pdf*) of $\pi(P)$ in all correct executions, and the *pdf* of $\pi(P)$ in failing executions: the bigger the "dissimilarity" between the two *pdf*s (according to a given similarity function; see [15] for details), the higher the bug relevance score of the predicate P.

Our approach, even using different information, shares with SOBER the principle of distinguishing correct and incorrect runs, and of evaluating the difference between them to assign a score to violations.

We applied SOBER in our case-study, comparing results with the ones obtained with *ReTool*. We instrumented test cases execution during regression tests, that is, in the step four of the procedure outlined in Section 4.1.1. The instrumented predicates have been then classified by the SOBER algorithm. Then, we proceeded with debugging considering the top 5 most suspicious predicates in the ranking, adhering to the choice made in [15] (i.e., $k = 5$, k being the number of predicates taken into account in the bug report).

Table 3: Results showing the percentage of code inspected before finding the bug with *ReTool* and SOBER.

Test	Inspected code (%) metric 1	Inspected code (%) metric 2	Inspected code (%) metric 3	Inspected code (%) SOBER
1	48.1	0.4	0.4	3.2
2	0.3	0.9	0.3	0.2
3	38.0	3.1	3.1	6.4
4	67.8	70.6	75.2	18.5
5	41.3	0.2	0.2	11.9
6	92.6	15.1	15.1	4.8
7	54.4	1.4	1.4	7.2
8	49.5	2.6	2.6	1.0
9	0.2	18.3	18.3	46.3
10	0.8	0.8	0.8	2.6
11	17.8	5.9	5.9	8.0
12	19.5	7.4	7.4	32.1
13	20.7	4.6	4.6	65.4
14	96.2	91.3	55.0	14.1
15	32.2	1.5	1.5	9.9
16	0.4	0.6	0.4	3.2
17	1.3	1.0	1.0	0.2
18	87.6	74.5	53.0	67.9
19	13.6	62.3	2.1	3.4
20	0.5	0.5	0.5	22.8
Mean	34.14	18.15	12.44	16.46

Table 3 shows obtained results in terms of (approximated) percentage of inspected statements to reach the bug. With respect to the three metrics of *ReTool*, we observe that SOBER has, in the average (i.e., last row of Table 3), better performances than the first metric of *ReTool* and similar performance to the second metric. The third metric of *ReTool* outperforms SOBER, even though with a little margin (it requires about 4% less code to be inspected with respect to SOBER), confirming that the choice of penalizing violations with lower bug relevance probability is valid. Note that experiments number 14 and 18 required a great percentage of code to be examined with *ReTool*, since the first violation was not bug related (i.e., it was a false positive). Hence, future effort should be devoted to eliminate the occurrence of false-positives. It should be also noted that SOBER and the third metric of *ReTool* exhibit a more regular behavior, that is, with low variations among experiments (with regard to this aspect, SOBER is slightly better than the third metric, since variance of SOBER results is about 10% less than variance of metric 3).

Figure 7 reports the same data grouped by the percentage of code inspected, that is, the number of bugs that have been found when a certain percentage of code is inspected. In this histogram, it is important to observe the number of bugs found with the lowest percentages of inspected code; indeed, bug reports requiring high percentages of code to be inspected may be not very useful.

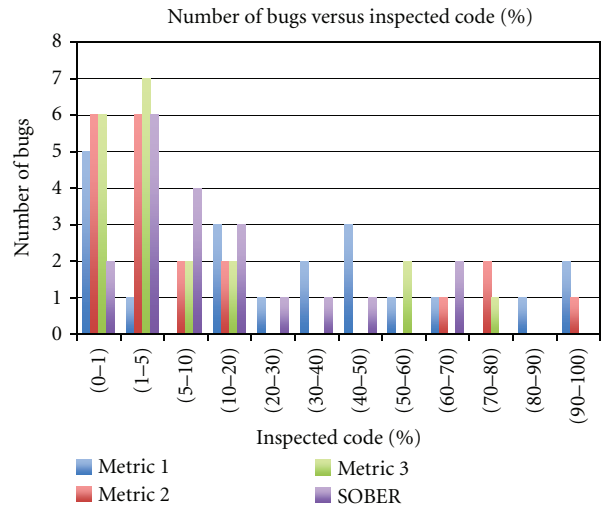


Figure 7: Number of violations (in the range 1–6) per fault type and per metric.

By inspecting less than 1% of the code, *ReTool* allowed finding more bugs than SOBER. Considering the inspection of at least 5% of the code (i.e., [0-1) and [1–5) intervals), SOBER outperforms metrics 1, but not metric 2 and 3; when the percentage goes up to 20% of inspected code, SOBER performance reaches metrics 2 and 3 (15, 16, and 17 bugs are found, resp., for SOBER, metrics 2 and 3, whereas metric 1 finds 9 bugs). The other bugs are clearly found when more than 20% of the code is inspected. It should be noted that metrics 2 and 3 do not have intermediate values: either they allow revealing a bug within 20% of code, or with more than 50% of code inspected.

Experimental results should be interpreted with care.

As threats to validity, it should be noted that:

(i) we selected for SOBER $k = 5$, as suggested by authors in [15]. Choosing a different value for k may change SOBER performances in these experiments. On the other hand, it makes no much sense trying different k values and then starting the debug; it would require too much time;

(ii) experimentation is carried out on one program; this, of course, limits the generalization of results (for instance, also performances of SOBER are different and are better with respect to experiments described in [15]). Certainly, the case study application affects experimental results;

(iii) the program structure may influence performance of *ReTool*. Indeed, violations indicate the method containing the bug, and also the variable(s), passed as argument, which violated the invariant. The inspection points are actually the set/get points regarding that variable(s) in that method. Thus, the average number of LoC of methods affect performance of *ReTool*: in our case, *JFreeChart* is well designed, since it has about 25 lines of code per method, hence actually reducing the inspection to few points per method, but this could not be always the case.

Considering that SOBER acts with a different source of information (i.e., predicates), neglecting information at method level, the latter point suggests that combining SOBER and *ReTool* is certainly a valuable way to be pursued in the future.

4.2. Case Study. Performances of the proposed technique (in terms of obtainable saving of time in the bug localization process) have been also experimentally evaluated on an application for multimedia library management, implemented by several student teams according to the TDD approach. Differently from the previous case, thus application is a newly developed application. The involved groups of students started from the same requirements, consisting in a set of 16 use cases describing the functionalities to be developed. The experimentation was carried out in two steps.

In the first step, the application development has conducted according to the TDD cycle. Each time a group of functionalities was developed, the team executed the corresponding regression tests, obtaining a certain number of failures. At the end of each of these cycles, the "faulty" version of the current implemented software was stored. Then the team started the debugging process and proceeded with the development of new functionalities.

In this first step, a manual debugging was carried out in order to locate and fix the bugs, during which information about the debugging process was collected. In particular, at each debugging session, developers recorded the number of fixed faults, the time needed to remove each fault, and information to identify a fault (i.e., the method where it was located and the code line number). At the end of the development of all functionalities, developers reported the list of manual debugging times for each located fault. Such results are shown in the third column of Table 4.

In the second step, the debugging times with the support of the tool are compared with manual debugging times previously measured. In particular, by using *ReTest*, the teams performed the debug on stored intermediate software faulty versions. To avoid mutual influences, each team worked on the program faulty versions developed by another team. Each time a failure was experienced, the tool reported violations rated according to their likelihood of being related to the bug, and debugging started from the first violation (the tool-supported debugging time is shown in the fourth column of Table 4).

Table 4 summarizes the results for three of the developer groups. In these experiments, violations were rated according to the best version of the implemented metric, that is, the third one. The table reports the manual debugging times (column 3), as well as the tool-supported debugging times (column 4). The first column represents the program version stored at the end of each implementation session. Since the groups developed the code independently, the number of times they decided to run regression tests (thus the number of implementation sessions) is also, in general, different from one group to another. Of course, the same stands for the number of committed faults (second column). The last row of each subtable reports the first-order statistics (mean and variance) for the manual debugging time (third column) versus the tool-supported debugging time (fourth column).

The fifth column values indicate if the tool rated in the first position a *bug-related* violation or a *false positive*. In particular, it reports the position of the first *bug-related* violation in the ranking carried out by the tool (a value per each fault). In the considered experiments, only one case presents a false positive in the first position (the last version of the second group).

The sixth column reports the length of the *violation chain*. As discussed, this parameter is more important than the fifth column parameter, since in all the cases, except one, the tool rated in the first position a *bug-related* violation. At the bottom of the table, mean and variance for the overall experiments, as well as the total debugging times, are reported.

Results show that performances of the tool allowed developers to gain, in the average, 2′52″ per fault (i.e., 40.75%) in the program developed by the first group, 3′48″ (i.e., 30.40%) in the one developed by the second group, and 4′41″ (i.e., 48.61%) in the third group's program. By averaging the whole set of experiments (last row), the mean saving time amounted to 3′54″ per fault (i.e., 40.91%). Variances of tool-supported debugging times also turned out to be lower in all the three experimented cases, being 5720″ (i.e., 15.02%) less than the manual debugging time in the first group program, 4380″ (i.e., 3.96%) in the second, and 3224″ (i.e., 3.80%) in the third (7222″, i.e., 8.80% in total).

A lower variance in results indicates that the use of a semiautomatic tool allows reducing the differences among faults in terms of difficulty to be found, hence balancing the effort required to find different kinds of faults.

Results are also supported by the t-student hypothesis test, which indicates that the difference between tool-supported debugging time and manual debugging time is significant at a confidence level greater than 98% (P value = .014774).

4.2.1. Considerations. As shown in the last columns of Table 4, when the first *bug-related* violation is not in the first position of the ranking (i.e., the tool rated a *false positive* as first violation), performances significantly reduce (see the last row of the second group), since the debugger spends too much time in the inspection of the code starting from a violation that is not related to the bug. Moreover, we experienced that in the other cases, in which the first rated violation is a *bug-related* violation, the tool-supported times are significantly lower than the manual times. In two cases, namely, the third row of the first group and the second row of the third group, the tool-supported time is higher than the manual time (in particular, when the violations chain is longer than eight). This happens because the debugger has to inspect too many code lines before locating the bug, due to the distance (in terms of number of violations) between the first-rated *bug-related* violation and the actual bug. The threshold of eight is of course limited to the experimented case study. In general, there will be for a given application a limit of the violation chain length over which the application of the tool is worse than the manual debugging. In the average, the usage of the tool allowed us to obtain significant time savings.

Table 4: Debugging times with manual versus tool-supported experiments. The 5th column indicates the position in the ranking of the first *bug-related* violation (BR) (a value per fault). The 6th column is the *VC length* in the execution trace from the bug activation to the bug-related violation V.

Group 1					
Version	No. faults	Manual time	Tool time	Position of the 1st BR	VC length
N1	1	11′10″	2′00″	1	1
N2	3	16′20″	9′30″	1,1,1	3,4,1
N3	1	7′40″	9′20″	1	9
		Manual time mean (and variance) per fault: 7′02″ (38070″)			
		Tool time mean (and variance) per fault: 4′10″ (32350″)			
Group 2					
Version	No. faults	Manual time	Tool time	Position of the 1st BR	VC length
N1	0	0	0	—	—
N2	2	23′40″	8′10″	1,1	3,1
N3	1	19′00″	12′50″	1	5
N4	1	7′20″	13′50″	2	1
		Manual time mean (and variance) per fault: 12′30″ (110470″)			
		Tool time mean (and variance) per fault: 8′42″ (106090″)			
Group 3					
Version	No. faults	Manual time	Tool time	Position of the 1st BR	VC length
N1	3	35′00″	9′30″	1,1,1	3,1,4
N2	1	8′10″	15′20″	1	14
N3	3	24′20″	9′50″	1,1,1	2,2,1
		Manual time mean (and variance) per fault: 9′38″ (84681″)			
		Tool time mean (and variance) per fault: 4′57″ (81457″)			
Summary					
		Manual time mean (and variance) per fault: 9′32″ (82087″)			
		Tool time mean (and variance) per fault: 5′38″ (74865″)			

4.2.2. Scalability. One more relevant point is the scalability of the proposed tool. The cost of our tool is caused by (i) instrumentation and (ii) ranking algorithm. As for the first point, the cost regarding instrumentation (i.e., *model builder* and *violation detector*) is dependent on Daikon tool suite, since it is the tool used to build invariants and detect violations. This aspect is treated in detail in [13]. To summarize the main points, this cost is (i) linear in the number of potential invariants at a program point; actually, since most invariants are soon discarded, time is linear in the number of true invariants, which is a small constant in practice; (ii) it is linear in the number of times a program point is executed (i.e., linear to test suite size), and (iii) linear in the number of instrumented program points (that is proportional to the size of the program). Some actions that can be taken to reduce instrumentation and invariants computation cost are as follows: (i) reducing the number of instrumented points to few critical points, (ii) reducing the number of executions, (iii) reducing the number of variables (see "the Daikon Invariant Detector", http://groups.csail.mit.edu/pag/daikon/).

The scalability of the ranking algorithm depends on the number of executions and the number of violations in each execution. Indeed, in order to assign probabilities to violations, for each violation v_i, the algorithm has to scan both correct and incorrect executions (let us denote them with N_f and N_c, resp.), to see if v_i is present in those executions, and to determine its position of occurrence (cf. with (2)). The time complexity per each violation is therefore $O((N_f + N_c) * AverageV)$, being $AverageV$ the average number of violations in each execution. Then, probability values assigned to violations have to be ordered. Thus, the total complexity is $O((N_f+N_c)*AverageV)+V* \log *V)$, where V is the final list of violations.

As for the second metric, the complexity is unchanged, since probability values for each method, which have to be summed to generate *M values* (cf. with (4)), are stored in a data structure and summed while the algorithm scrolls and scores violations.

As for the third metric, a further operation proportional to the number of the final list of violations V is required, in order to divide p_i values for the relative position of the violation, hence getting to $O((N_f+N_c) * AverageV)+V * \log * V + V)$.

Comparing with SOBER, we note that (i) the first and the second metrics have the same complexity, being SOBER

complexity equal to $O((N_f + N_c) * k) + k * \log * k)$ where k is the number of predicates; (ii) the third metric has higher complexity than SOBER, due to the additional factor V; and (iii) the number of violations V is typically lower than the number of predicates, since it refers to methods.

5. Related Work

In the past, much work dealt with bug localization by either static analysis or dynamic analysis techniques. Static analysis approaches rely on source code knowledge and are used to verify program correctness against one or more properties (e.g., relevant examples are software model checking [25, 26], or static slicing, by Weiser [27, 28]).

Dynamic analysis, as opposed to static analysis, aims to give information about the system by observing its execution traces. It attempts to overcome the static analysis limitations, such as the difficulty to cope with large-size applications, with changes of evolving systems, and with the increasing use of off-the-shelf (OTS) items, where the source code is not always available. Most of approaches today are classifiable as dynamic analysis; examples are (i) dynamic slicing, (ii) techniques based on behavioral models, (iii) and statistical debug. Our approach also falls into dynamic analysis category, being based on runtime execution traces analysis.

Dynamic slicing approaches monitor the system during an execution and trace the program elements covered. Only the statements covered at runtime are considered to build the slices; this lead to smaller slices with respect to those generated by static slicing approaches. It was first introduced by Korel and Laski [29]. A recent approach based on dynamic slicing is proposed in [30], which through a *callstack-sensitive* slicing and slices intersection reduces the slice sizes by leveraging the series of calls active when a program fails. Similar approaches considering the sequence of *active* function calls to improve slicing were previously proposed in [31, 32].

Several dynamic analysis approaches and tools based on behavioral models appeared in the literature (e.g., [13, 33–36]). Among these, one of the most successful ones is *Daikon* [13]. It is a tool that automatically discovers likely invariants from executions of instrumented programs, by building a wide spectrum of invariants using definite invariant templates. The authors in [33, 37] extend *Daikon*, proposing a tool (BCT) that allows to infer invariants on both the observation of call sequences among monitored objects (i.e., interaction invariants), and on the exchanged parameter values (i.e., the I/O invariants). It uses the *Daikon* inference engine for discovering I/O invariants, and an incremental algorithm, named *Kbehavior*, to build a finite state automata (FSA) representing the call sequences among objects.

The DIDUCE project tool [36] tests a more restricted set of predicates within the target program and relaxes them in a similar manner to *Daikon* at runtime. When the set of predicates becomes stable, it relates further violations as indications of potential bugs.

Also our approach relies on the use of behavioral models builder tools to describe the behavior of the system under development, and in particular it is based on *Daikon*. However, differently from the mentioned tools and techniques, the presented tool adds the ability to distinguish those behaviors most likely related to the bug, through a ranking process of detected violations by the implemented proximity metrics.

There are several approaches that use statistical considerations to assign a score to events of interest. One technique falling into this category is SOBER [15], presented in Section 4.1.5. Even if SOBER describes the application behavior differently from our approach (it uses predicates), our tool takes some concepts from it to define the metric for violations ranking (such as the distinction between correct and incorrect executions to judge a violation).

Other statistical techniques have been proposed in the program analysis and bug detection [16, 17, 38]. Among these, authors in [17] present a statistical debugging algorithm that aims at separating the effects of different bugs and identifies predictors that are associated with individual bugs, with the goal of making it easier to prioritize debugging efforts. Statistical debugging is recently improved by an adaptive monitoring strategy [39] that starts by monitoring a small portion of the program, and then automatically refines instrumentation over time. Based on analysis of feedback in a given stage, the technique automatically chooses new behaviors that could cause failures and monitors them during the successive stage, thus reducing the overhead. This approach can be complementary to ours, since it is able to improve performances in terms of instrumentation overhead.

Similarly, the HOLMES project [18] also contributed to statistical debugging by introducing a strategy based on paths profiling (it counts the number of times each path is taken in an acyclic region) to pinpoint likely causes of failures. The tool also uses an iterative profiling for an adaptive instrumentation, by exploiting partial feedback data to select and instrument predicates in successive iterations.

The authors in [19] propose a context-aware approach also based on control flow paths, which considers predicate correlations and control flow paths that connect the bug predictors for better diagnosis of bugs. One more statistical debugging algorithm is presented in [20], which aims at identifying predictors that are associated with individual bugs, separating the effects of different bugs. The algorithm, differently from previous ones (and from our approach), does not consider exact values of predicate counts but considers whether a predicate was true at least once, making no further distinctions among nonzero counts. This makes instrumentation lighter and more scalable. However, unlike in [39], the instrumentation is nonadaptive.

Another relevant tool falling in this category is Tarantula [21]. Tarantula algorithm assigns scores to statements and ranks the statements from most suspicious to least suspicious. It utilizes a concept similar to ours, in that it considers pass/fail information about test cases (along with the entities that were executed by each test case, such as statements, branches, methods, and the source code), and the concept that entities executed by failed test cases are more likely to be

faulty than those executed by passed test cases. Performances of this tool are also evaluated in [40].

An important debugging technique is the delta debugging [41, 42]. It was introduced in 1999 by Zeller in a seminal paper in this area [43] and used in several contexts (e.g., in diagnosis [44]). Delta debugging aims at simplifying or isolating failure causes by systematically narrowing down failure-inducing circumstances until a minimal set remains.

The algorithm aims at identifying the smallest set of changes that caused a regression in a program. It requires the following information: a test that fails because of a regression, a previous correct version of the program, and the set of changes performed by programmers. The delta debugging algorithm works in this way: it iteratively applies different subsets of the changes to the original program to create different versions of the program and identify the failing ones. The minimal set of changes that permit to reproduce the failure is reported.

Our approach lies in between "*Daikon-like*" techniques, that is, based on behavioral models and statistical debug techniques, since it uses the former approach to describe the application behavior (in terms of invariants) and to detect violations (in the training and detection phases), and some criteria of the latter to rate those anomalous behaviors possibly related to bugs (in the ranking phase).

Moreover, a key difference is that the presented technique is specifically tailored for the TDD process; the continuous iterations of TDD allowed us to embed a semiautomatic debug step from the earliest stages of the development cycle, where a greater effectiveness in bug detection is actually required. Indeed, debugging applications during their development help greatly reducing maintenance costs.

The structure of the TDD approach also justifies the adoption of techniques based on behavioral models in the training phase, since repeated iterations allow for building more and more accurate behavioral models.

Finally, it is worth reporting some different approaches to debugging, such as those taken in [45] and in [46]. In the former, authors present a technique for supporting in-house debugging of field failures, based on a three-step procedure: field failing executions recording, execution traces minimization, and replaying of the minimized execution to help debugging (within a debugger). In the latter ([46]), a spectra-based technique is presented; here, the concept of execution time is considered as an indicator of anomalous behaviors, and hence of potential bugs: time spectra are collected from passing and failing runs, observed behavior models are created using the time spectra of passing runs, and deviations from these models in failing runs are identified and scored as potential causes of failures.

The tool presented in [47], named *Whyline*, combines more program analysis techniques, such as static and dynamic slicing, precise call graph, in order to allow user to choose a set of *why* or *why didn't* question about program output derived from program's code and execution, and then to generate an answer to the questions. Such questions represent the user's will to understand the program's behaviour starting from the output.

There are several other, more narrowed, approaches that focus on some specific aspects of bug localization problem. Examples are the techniques presented in [48, 49], where the problem of memory leak detection is faced, and in [50], whose focus is on faults in concurrent programs.

6. Conclusion and Future Work

In this work, we presented a technique to help developers localize bugs, embedded in the test-driven development cycle. The technique has been implemented in a tool, written in Java, that during the unit tests execution allows to build a model of the expected behavior of the software under development and then compares the behavior observed in the regression test phase with the built model in order to detect anomalies.

These anomalies are rated according to the defined metrics and assigned a probability to be related to the failure-causing bug. The developer uses the ranking to inspect the code and localize the bug. The proposed metrics have been evaluated through an existing open-source application, in which a fault injection campaign has been carried out and the goodness of the ranking process carried out by the tool has been analyzed. The technique performances have been verified by evaluating the tool's ability to reduce debugging time with respect to manual debugging on a new application developed by groups of students according to the TDD cycle. Results encourage further investigation of the tool on other software applications and potential integration with existing approaches.

In particular, as for future work, we are attempting to improve the accuracy of the implemented metrics in the identification of bug location, by considering the combination of features of other statistical-based algorithms with our technique. The goal is to improve the bug proximity information inside the method that most likely contains the bug, by combining violations of invariants on methods and on object values with approaches based on predicates, which currently do not consider information related to methods.

Moreover, we are evaluating the introduction of static information derived from software specifications (such as constraints on some critical exchanged values or mandatory sequences of calls) in order to ease the distinction among *false positives* and erroneous behaviors.

Finally, we are investigating the factors that mainly impact the value of α and β, in order to reduce the start-up time spent to tune them; preliminary analyses show that the coupling degree among modules has a significant impact, but further investigations are needed.

References

[1] D. Janzen and H. Saiedian, "Test-driven development: concepts, taxonomy, and future direction," *Computer*, vol. 38, no. 9, pp. 43–50, 2005.

[2] R. Kaufmann and D. Janzen, "Implications of test-driven development: a pilot study," in *Proceedings of the 18th ACM*

SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '03), pp. 298–299, Anaheim, Calif, USA, 2003.

[3] E. M. Maximilien and L. Williams, "Assessing test-driven development at IBM," in *Proceedings of the 25th IEEE International Conference on Software Engineering (ICSE '03)*, pp. 564–569, IEEE CS Press, May 2003.

[4] A. Gupta and P. Jalote, "An experimental evaluation of the effectiveness and efficiency of the test driven development," in *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM '07)*, pp. 285–294, September 2007.

[5] R. C. Martin, "Professionalism and test-driven development," *IEEE Software*, vol. 24, no. 3, pp. 32–36, 2007.

[6] B. George and L. Williams, "A structured experiment of test-driven development," *Information and Software Technology*, vol. 46, no. 5, pp. 337–342, 2004.

[7] L. Williams, E. M. Maximilien, and M. Vouk, "Test-driven development as a defect-reduction practice," in *Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE '03)*, p. 34, IEEE Computer Society, 2003.

[8] B. George and L. Williams, "An initial investigation of test driven development in industry," in *Proceedings of the ACM Symposium on Applied Computing (SAC '03)*, pp. 1135–1139, ACM Press, 2003.

[9] T. Bhat and N. Nagappan, "Evaluating the efficacy of test-driven development: industrial case studies," in *Proceedings of the 5th ACM/IEEE International Symposium on Empirical Software Engineering (ISESE '06)*, pp. 356–363, ACM Press, 2006.

[10] A. Geras, M. Smith, and J. Miller, "A prototype empirical evaluation of test driven development," in *Proceedings of the 10th International Symposium on Software Metrics (METRICS '04)*, pp. 405–416, IEEE Computer Society, 2004.

[11] M. M. Müller and O. Hagner, "Experiment about test-first programming," in *Proceedings of the International Conference of Software Engineering (ICSE '02)*, vol. 149, no. 5, pp. 131–136, 2002.

[12] M. Pancur, M. Ciglaric, M. Trampus, and T. Vidmar, " Towards empirical evaluation of test-driven development in a university environment," in *Proceedings of the International Conference on Computer as a Tool (EUROCON '03)*, vol. 2, pp. 83–86, 2003.

[13] 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.

[14] M. D. Ernst, J. H. Perkins, P. J. Guo et al., "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 35–45, 2007.

[15] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: statistical model-based bug localization," in *Proceedings of the 10th European Software Engineering Conference (ESEC/FSE '05)*, pp. 286–295, 2005.

[16] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles," in *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*, pp. 339–348, 2001.

[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pp. 15–26, 2005.

[18] T. M. Chilimbi, B. Liblit, K. K. Mehra, A. V. Nori, and K. Vaswani, "HOLMES: effective statistical debugging via efficient path profiling," in *Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE '09)*, pp. 34–44, IEEE Computer Society, 2009.

[19] L. Jiang and Z. Su, "Context-aware statistical debugging: from bug predictors to faulty control flow paths," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds., pp. 184–193, ACM Press, 2007.

[20] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, V. Sarkar and M. W. Hall, Eds., pp. 15–26, ACM Press, 2005.

[21] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*, D. F. Redmiles, T. Ellman, and A. Zisman, Eds., pp. 273–282, ACM Press, 2005.

[22] M. Hiller, J. Christmansson, and M. Rimèn, "An experimental comparison of fault and error injection," in *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE '98)*, pp. 369–378, 1998.

[23] R. Chillarege, I. S. Bhandari, J. K. Chaar et al., "Orthogonal defect classification—a concept for in-process measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943–956, 1992.

[24] J. Durães and H. Madeira, "Emulation of software faults: a field data study and a practical approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 849–867, 2006.

[25] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 1999.

[26] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: localizing errors in counterexample traces," in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*, pp. 97–105, 2003.

[27] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[28] M. Weiser, "Programmers use slicing when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.

[29] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[30] S. Horwitz, B. Liblit, and M. Polishchuk, "Better debugging via output tracing and callstack-sensitive slicing," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 7–19, 2010.

[31] D. Binkley, "Semantics guided regression test cost reduction," *IEEE Transactions on Software Engineering*, vol. 23, no. 8, pp. 498–516, 1997.

[32] J. Krinke, "Context-sensitivity matters, but context does not," in *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '04)*, pp. 29–35, 2004.

[33] L. Mariani and M. Pezzè, "Behavior capture and test: automated analysis of component integration," in *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '05)*, pp. 292–301, 2005.

[34] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, "Mining object behavior with ADABU," in *Proceedings of the International Conference on Software Engineering (ICSE '06)*, pp. 17–24, 2006.

[35] B. Schmerl, D. Garlan, and H. Yan, "Dinamically discovering architectures with DiscoTect," in *Proceedings of the European Conference on Software Engineering (ESEC/FSE '05)*, pp. 103–106, 2005.

[36] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pp. 291–301, 2002.

[37] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pp. 501–510, ACM Press, New York, NY, USA, 2008.

[38] A. Podgurski, D. Leon, P. Francis et al., "Automated support for classifying software failure reports," in *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pp. 465–475, 2003.

[39] M. B. Dwyer, A. Kinneer, and S. G. Elbaum, "Adaptive online program analysis," in *Proceedings of the 29th IEEE International Conference on Software Engineering (ICSE '07)*, pp. 220–229, IEEE Computer Society, 2007.

[40] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE '09)*, pp. 56–66, IEEE Computer Society, 2009.

[41] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[42] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pp. 342–351, ACM Press, New York, NY, USA, 2005.

[43] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '07)*, pp. 253–267, Springer, London, UK, 2007.

[44] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou, "Triage: diagnosing production run failures at the user's site," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, pp. 131–144, ACM Press, 2007.

[45] J. Clause and A. Orso, "A technique for enabling and supporting debugging of field failures," in *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pp. 261–270, IEEE Computer Society, 2007.

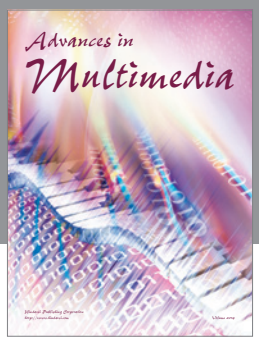
[46] C. Yilmaz, A. M. Paradkar, and C. Williams, "Time will tell: fault localization using time spectra," in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, W. Schafer, M. B. Dwyer, and V. Gruhn, Eds., pp. 81–90, ACM Press, 2008.

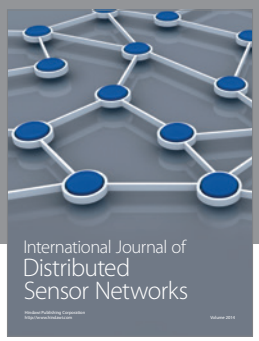
[47] A. J. Ko and B. A. Myers, "Debugging reinvented: asking and answering why and why not questions about program behavior," in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pp. 301–310, ACM Press, New York, NY, USA, 2008.

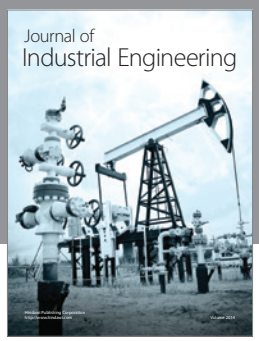
[48] G. Xu and A. Rountev, "Precise memory leak detection for java software using container profiling," in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pp. 151–160, ACM Press, New York, NY, USA, 2008.

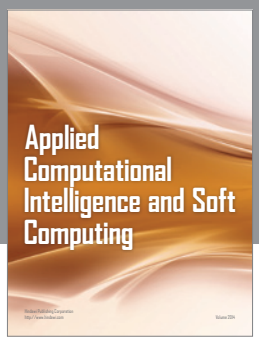
[49] J. Clause and A. Orso, "LEAKPOINT: pinpointing the causes of memory leaks," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, vol. 1, pp. 515–552, ACM Press, New York, NY, USA, 2010.

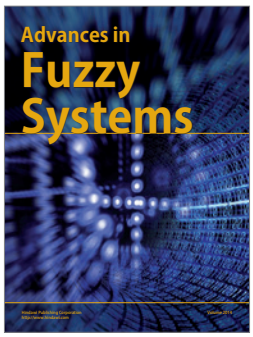
[50] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: fault localization in concurrent programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, vol. 1, pp. 245–254, ACM Press, New York, NY, USA, 2010.

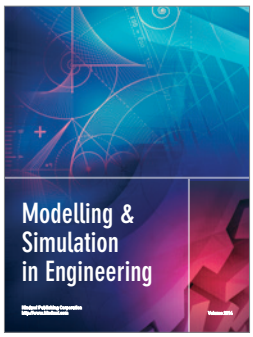


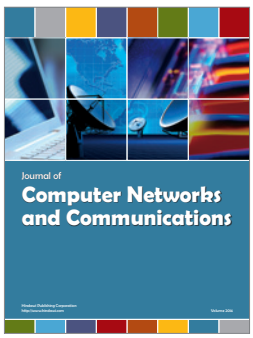












Hindawi
Submit your manuscripts at
http://www.hindawi.com