EFFECTIVE FAULT LOCALIZATION FOR SOFTWARE TESTING AND DEBUGGING

Mario Gonzalez Macedo

Ortask
mgm@ortask.com

ABSTRACT

The explosion of complexity in modern software systems means that, once practitioners of software testing have detected a fault[1] [2], they have to spend increasingly more time and work localizing[3] such faults in order to provide reliable and recreatable information to help fix such faults. Another side-effect of this complexity is that a few failing tests might cause practitioners to question whether there is still any value in the output of any remaining passing tests due to the belief that erroneous states from failing tests have cascaded to passing ones. Such uncertainty is all the more poignant in novice practitioners that are still learning the science, art and craft of proper software testing or debugging.

This paper contributes several important results to software testing and debugging in the form of an effective fault localization methodology. The methodology includes: 1) a novel utilization of a conceptual device called reachability graphs; 2) a new result in the form of a theorem that allows the unmasking of irrelevant variables to reduce the search space; and 3) a celebrated and useful result form causality theory to assist in handling other scenarios.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; K.7.3 [**The Computing Profession**]: Testing, Certification, and Licensing

General Terms

Measurement, Reliability, Theory

Keywords

Fault localization, Empirical software engineering, Software testing

1. INTRODUCTION

One of the goals of software testing is to produce evidence that a piece of software works as intended [1, 2]. That evidence might be in the form of hard, deterministic data or even stochastic results that statistically show the correctness of the software system to some degree, among others. However, with modern software systems becoming increasingly complex, it can be difficult and time-consuming to narrow down the exact location of a detected fault. Fault localization is important for several reasons, including recreating the fault, fixing the fault, as well as creating appropriate documentation when necessary[4]. Fault localization is all the more important when applying manual testing methodologies because automation might not be present to easily record (or script) and replay the steps necessary to recreate previously encountered faults[5]. It is therefore obvious that all forms of software testing (and development) benefit greatly from effective and fast fault localization methods.

An additional and related concern shared by many practitioners of software testing (including developers) is whether their tests can be trusted once a few tests have failed within the same execution. In other words, there is skepticism from many practitioners – especially novice – regarding whether their remaining tests are indeed "telling them the truth" about their systems under test, once a few failed tests have been observed. This is due to the belief that faulty states from failing tests somehow carry or cascade to passing ones without practitioners being able to determine exactly where those points of cascade happened. This perception is best summarized by the following statement from one such practitioner:

When a tester completes a test, there is some kind of an expected result or post-condition. There is also an unwritten post-condition. It is: "Nothing else that could threaten the value of this product happened." It is impossible to know if that post-condition is met, and even if we encounter a problem down the line due to a memory leak, or extraneous data in the database, or corrupted files, then we will not know which test it was that "failed" (personal communication from Joe Harter, April 8, 2013).

Yet, even with this apparent limitation, veteran practitioners of software testing are able to use their skills to successfully find such points of failure in software components. With respect to tests, such information is then utilized to localize and distinguish faults for failing and passing tests. In turn, this information allows them to determine which state carried to passing tests, if any, and which tests need to be re-run.

This paper provides a methodology by which such veteran practitioners are able to successfully localize faults in their SUT[6] – and, likewise, in their tests – with a high degree of effectiveness.

2. THE UNDERLYING PROBLEM OF FAULT LOCALIZATION

The underlying problem for fault localization is a phenomenon called *confounding* (also called *confounding bias*) [7]. Therefore, we must first understand what confounding bias means.

In simplest terms, confounding is "an unknown causal quantity that is not directly measurable from observed data." [7] In other words, an effect is confounded when its cause (either actual or contributory) is not clear from measurements, experiments and data gathered.

[1] This paper uses the accepted definition for *fault* as stated in [12]
[2] Also informally called a *bug*.
[3] Also informally known as fault *isolation*.
[4] For instance, when such faults cannot be fixed.
[5] Such is a mayor limitation of exploratory testing methods because they aim to loosely describe what to test (say as charters), rather than specifically prescribe both what and how to test.
[6] ***S***ystem ***U***nder ***T***est

The difficulties encountered by naive fault localization methods in the face of confounding bias can be understood and explained when using Pearl's mathematical notation for confounding and probabilistic association as follows [7]:

> DEFINITION 1. ***Associational No-Confounding***
>
> *Let X and Y be variables of interest that we suspect have some causal relationship (particularly that X might cause Y). Further, let T be the set of variables that are unaffected by X.*
>
> *We say that X and Y are not confounded in the presence of T if each member $Z \in T$ satisfies at least one of the following conditions:*
>
> $C_1 : P(X|Z) = P(X)$
>
> $C_2 : P(Y|Z, X) = P(Y|X)$

In words, C_1 states that when $Z \in T$ is not associated with *X*, then the probability of *X* given that we observe *Z* is simply the probability of observing *X* alone (i.e. *X* and *Z* are independent because observing *Z* does not affect the occurrence of *X*, and vice versa). C_2 states that when we observe both *X* and *Z*, and *Z* is not associated with *Y* (given *X*), then the probability of observing *Y* equals that probability of *Y* given that we only observe *X*. If any one of the conditions above is satisfied by each variable in *T*, then *X* and *Y* are *not* confounded and we can be sure that *X* indeed causes *Y*. Otherwise, the causal relationship between *X* and *Y* is unclear.

Even though the associational definition above sets forth clear conditions for when two variables of interest are not confounded, there is a particular implied assumption that presents difficulty for fault localization for software in general and software testing in particular. Note that *T* is not simply any set of variables that are unaffected by *X*. Rather, it is the set of *all* such variables – and therein lies the difficulty with naive fault localization methods. In other words, unless we can guarantee that each and every variable (say, a component, module, class, function, or any other logical piece of a particular software system in question) that is not affected by *X* satisfies C_1 or C_2, we cannot be completely sure that *X* indeed has a causal relationship with *Y*. Moreover, even the relationships between *X* and every possible *Z* might be unclear to us, therefore limiting the accuracy of our model for the SUT and, hence, our testing.

Consider an example to illustrate the limitation of utilizing naive fault localization methods in the face of confounding. Assume that *Y* represents "*Result of authenticating to an application's GUI*[7]" and *X* stands for "*Account status*". We may think that if we get an authentication error ($Y = \bar{y}$) when entering our credentials in the application's GUI, it is because we suspect our password is wrong or possibly expired ($X = \bar{x}$). Yet there might be a variable *Z* that we do not know about, affecting *X* and *Y*, thus confounding the causal relationship. For instance, *Z* might be a newly introduced authentication module that returns incorrect results due to a bug even though the given password might be valid. A different scenario might be that we are unable to authenticate due to another variable *Z'* (representing "*System hacked status*") whereby the logic of the application has been compromised by a malicious hacker through code injection techniques. Yet another scenario contains both *Z* and *Z'*, whereby the software system has been hacked and the authentication module is also faulty.

As these examples illustrate, the effectiveness of naive fault localization methods given confounding bias are directly related to how we define our models for a SUT. However, the way we do so is inherently limited by Closed-World assumptions, marginality[8], barren proxies[9], and incidental confoundedness that lead to false negatives as well as false positives [7]. Our inability to know for certain all the states and conditions under which a software system executes restricts our knowledge of the precise causal relationships and, hence, root cause of a given behavior of a SUT. For instance, even though our initial suspicion might have been that *X* causes *Y*, our not knowing about variables *Z* and *Z'* confound the suspected causal relationships and prevent us from determining the root cause.

Given the disappointing realization about our impediment to fully comprehend the complete mechanisms by which a given SUT works – and the apparent severe limitations of testing as implied by the difficulties to fault localization – why is it that the practice and profession of software testing is still utilized around the world as a mayor and highly accurate way to ascertain the degree of quality for software? To answer this question, we must delve deeper into how effective fault localization works.

3. HOW EFFECTIVE FAULT LOCALIZATION WORKS, PART 1

While it is true that it is difficult to know the complete set of states of any software system or even to know the current (complete) state transition of a particular SUT at a given time, it is *not* true that we can never isolate *any* causal relationships in a SUT to a high degree of certainty. Moreover, we can further use this causal information – along with the scientific method – to guide us to the actual causal relationships. In other words, even with the inherent limitations hindering our knowledge of the underlying workings of a software system, we can still trust our tests in many (if not most) cases with the information they provide to us about the SUT. In turn, this implies that we need not worry about entering permanent scenarios where we do not know *"which test it was that 'failed'"*.

The reason for this is because the unwritten and unobservable post-condition ("*Nothing else that could threaten the value of this product happened*") is fundamentally at odds with the foundation of software testing – the scientific method – which has allowed all engineering and scientific fields to flourish. While anything can indeed happen that "*could threaten the value of this product*", the key observation is that not everything can happen that would threaten the value of a product *with respect to the intent of each particular test and its set of oracles* [29]. Therefore, once we start targeting pieces of any software system, we can begin to *isolate* relationships that may be at work for those pieces and then gradually rule out *irrelevant* variables that we originally might have considered important. Once those pieces are understood and irrelevant variables thrown out of our model, we can gradually move our view of the SUT to higher levels, all the while remaining certain that the relationships we found before are not disturbed simply by including more of the SUT in our model.

The following two axioms illuminate the foundation that helps us understand how this is done.

[7] ***G***raphical ***U***ser ***I***nterface

[8] Association between *X* and multiple variables in *T*.

[9] Including irrelevant variables in the model that are associated to relevant ones.

AXIOM 1. ***System Behavior-State Mapping***

Any behavior of a system can be mapped to some sequence of states of the system.

AXIOM 2. ***Monotonic Knowledge of System Behavior***

To an ideal knowledge base, the amount of knowledge about a system's behavior never decreases and is never contradictory.

That is, let KB_{t_1} represent a fact about the SUT at time t_1. Further, let KB_{t_2} represent a fact about the SUT at some future time t_2. Then the following two conditions hold:

$M_1 : |KB_{t_2}| \geq |KB_{t_1}| \quad \forall t_1, t_2 \in TIME \ni t_2 \geq t_1$

M_2 : if $q \in KB_{t_i}$, then $\bar{q} \notin KB_{t_i}$, where q is a fact about the SUT at any time t_i

For instance, suppose that at time t_1 we have that $KB_{t_1} = \{q_1, q_2\}$, where each q_i represent a piece of knowledge about the SUT such as causal relationships or associations between components[10]. Also suppose that at some later time t_2, we find out that $\bar{q_1}$ (i.e. that the piece of information we knew as q_1 is not true[11]). Then axiom 2 states that $KB_{t_2} = \{\bar{q_1}, q_2\}$, and not simply $\{q_2\}$, and certainly not $\{q_1, \bar{q_1}, q_2\}$ as that would entail a logical contradiction about the behavior of the SUT.

Given these two axioms, we then need to understand the SUT in terms of *abstract states*

DEFINITION 2. ***Abstract State***

An abstract state is a finite set of properties about the SUT that are expected to be true at a specific point in a test. Further, an abstract state is only composed of properties that are relevant at that particular point of the test.

All abstract states are members of the class $ABSTRACT\_STATE_{SUT}$, which is the entire collection of sets of properties that the given SUT is expected to have.

Finally, an abstract state represents a collection and an abstraction of one or more underlying (i.e. real) states of the SUT.

Note that by the definition above, no irrelevant property forms part of an abstract state. For instance, time is always a part of real system states. However, unless time is relevant to a particular point in a test (either because we know it is so, or because we suspect it is so), the associated abstract state will not contain any property related to time. Due to this "casting away" of irrelevant details, abstract states can represent one or more real states from the SUT.

At this point, there might be some confusion about what the utility of abstract states is (and even about the definition itself) given they are only to contain *relevant* properties. To answer this, note that constructing abstract states does not automatically imply that we must only include the actual properties that are relevant to the SUT – indeed, that is the goal we are striving to achieve. We can certainly introduce properties to any abstract state that we *suspect* to be relevant for our model, even though those properties might not be at all relevant in reality.

Abstract states allow us to reason about the workings of a given SUT without over-complicating our (mental) model. They also allow us to use the scientific method to rule out faulty assumptions, thus increasing the effectiveness of our fault localization method. For instance, using our example from before, if we believe that the state of our account ($X = \bar{x}$) is related to the password validation error ($Y = \bar{y}$), then we can place $\bar{x}$ and $\bar{y}$ as properties in their own abstract states connected with a *reachability* edge. Figure 1 illustrates these abstract states using a *reachability graph*[12].

Figure 1: A reachability graph showing state reachability between two separate abstract states, with each abstract state containing a relevant property.

3.1. Isolation of Irrelevant Abstract State Properties

Given definition 2 for abstract states and the axioms of our conceptual framework, we are ready for the following theorem which enables the first step toward helping us distinguish between relevant and irrelevant properties in a SUT. This, in turn, lays the first part of the methodology for effective fault localization.

THEOREM 1. ***Isolation of Irrelevant Abstract State Properties***

Let p be some arbitrary property about a given SUT. Further, let $as, as', as'' \in ABSTRACT\_STATE_{SUT}$ be abstract states such that $p \in as'$ and $\bar{p} \in as''$.

p is relevant to as (written $p \sim as$) if and only if there exists a reachability path from either as' or as'' (but not both) to as. Otherwise, p is not relevant to as (written $p \not\sim as$).

Proof part 1 ($\rightarrow$). If p is relevant to as, then there exists a path from either as' or as'' (but not both) to as. Assume to the contrary that there exists a reachability path from as' and as'' to as, as in figure 2.

[10]For example, whether an application's database influences the application's middleware.

[11]This can happen when we gather new information about the SUT that corrects faulty or previous knowledge.

[12]A reachability graph for abstract states is an undirected graph where edges connect abstract states that are known or suspected to be related. The edges can be labeled with timestamps to provide further granularity for event sequences, indicating that reachability is time-wise monotonically increasing (i.e. reachability never goes backward in time). They are called reachability graphs from the fact that when two abstract states are related, one abstract state can *reach* the other via state transitions in the underlying SUT (i.e. the reflexive transitive closure of states.)

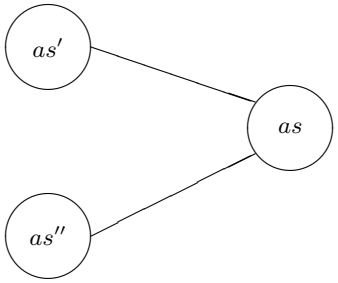


Figure 2

Suppose we abstract away all properties (including other relevant properties) from both as' and as'' except p and $\bar{p}$. Denote the new abstract states $\hat{as'}$ and $\hat{as''}$, respectively, as figure 3 shows.

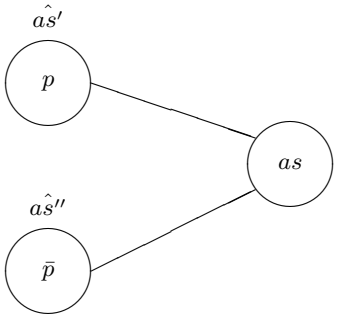


Figure 3

Let us now introduce an irrelevant property, r, into both $\hat{as'}$ and $\hat{as''}$ as in in figure 4.

Note that we can then replace the first conjunct in each abstract state by $p' = \bar{p} \vee p$. This is shown in figure 5.

Since p' is equivalent to *true*, we can do away with p' and leave only r in each abstract state, as in figure 6.

However, note that doing so makes p irrelevant, contradicting our assumption that p is relevant. ♦

Proof part 2 (←). If p is irrelevant to as, then its value does not affect the connectivity of its containing abstract state to as. Assume to the contrary that its value does indeed affect the connectivity of its containing abstract state to as. Then changing the value of p in existing abstract states will either induce or remove an edge, thus changing reachability (so that either a path is created through that abstract state or a path is removed, respectively). This is illustrated in figure 7.

We can introduce any other irrelevant property into the abstract states while keeping the edges unmodified. We do this by introducing the same irrelevant property to both abstract states in question. Let r be the newly introduced irrelevant property as figure 8 illustrates.

We then change the value of r in all abstract states, thus changing the edges from those abstract states to as. Figure 9 shows this change.

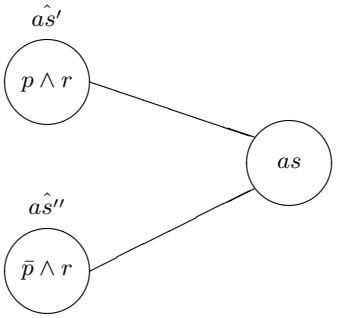


Figure 4

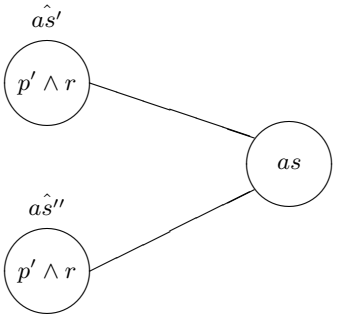


Figure 5

Since r is irrelevant to as, we can take it out of the abstract states. The resulting abstract states, however, are the original ones except that the value of their property p has changed without changing their edge to as, which contradicts our assumption. Figure 10 shows the resulting configuration of abstract states, illustrating the contradiction. ♦

As a pattern for ruling out irrelevant properties, theorem 1 is a *necessary* criterion in the sense that it does not lead to false positives. That is, it does not declare a property P as relevant to an abstract state when in fact it is not. This is summarized in the following corollary.

COROLLARY 1. ***Theorem 1 Yields No False Positives***

Theorem 1 is a necessary condition to rule out irrelevant properties as it never labels irrelevant properties as relevant.

3.2. Applications of Theorem 1

The following examples[13] illustrate the utility and application of theorem 1 for effective fault localization. Although very different in nature and in circumstances, several of these examples reduce to the same pattern once illuminated by the use of theorem 1.

[13] Even though no names are disclosed in these examples, they are taken directly from real cases from the authors' own portfolio with clients and customers.

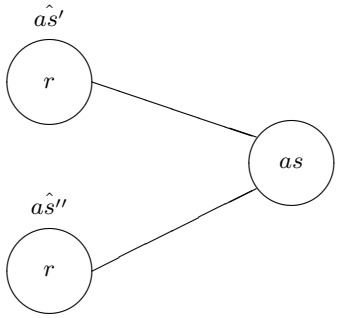


Figure 6

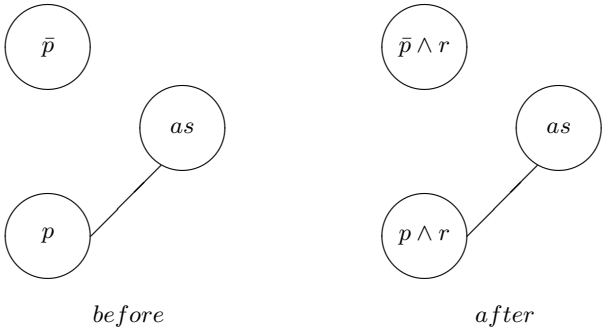


Figure 8

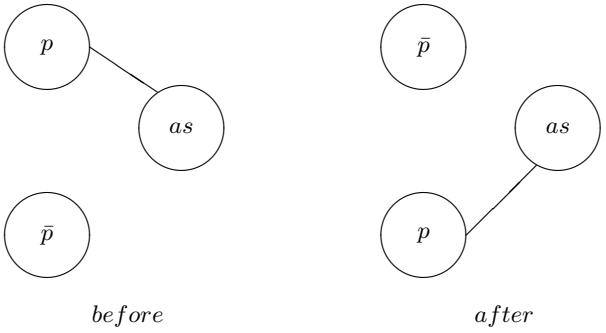


Figure 7

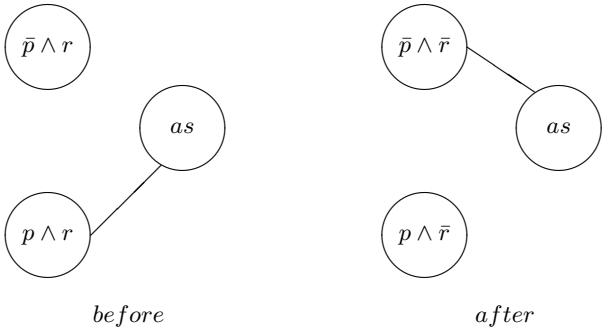


Figure 9

3.2.1. Login and LDAP integration

While testing the integration of LDAP[14] with the authentication capabilities of an enterprise storage monitoring application, it was noticed that authentication to the application succeeded using local OS accounts prior to integrating LDAP, but then failed immediately after configuring it for LDAP. Further, reverting the changes to the application (by disabling the application's LDAP integration) still resulted in login failures using those same local OS accounts.

The customer hypothesized that enabling LDAP integration somehow caused irreversible damage to the application, which prevented local OS authentication.

When modeled using abstract states and reachability graphs, the problem can be understood as follows. Let A represent the authentication result, and let L denote the state of LDAP configuration in the application (i.e. $\bar{l}$ means LDAP is disabled, and a means that authentication is possible.) Initially (at time t_0), we have LDAP integration not enabled, $\bar{l}$, and successful authentication of local OS accounts, a, to the application. At time t_1 we enable LDAP integration, l, and we notice that authentication to the application with local OS accounts now fails, $\bar{a}$. Then at time t_2, we disable LDAP integration again, $\bar{l}$, but still get authentication failures $\bar{a}$. This is shown in figure 11.

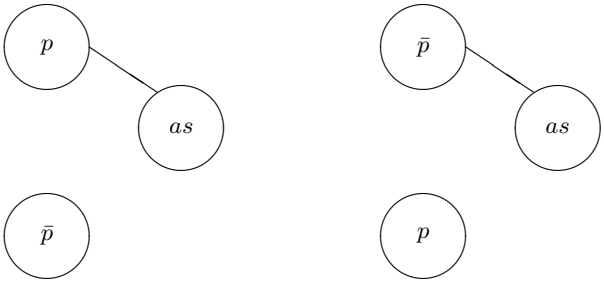
Figure 12 shows a reachability graph with a compressed progression of (abstract) states. From this new reachability graph and theorem 1, we can quickly acquire the insight that property $\bar{l}$ is irrelevant to A (the authentication result). Likewise, and perhaps more telling, L (the status of the application's LDAP integration) is irrelevant to property $\bar{a}$. Indeed, the inability to authenticate to the application turned out to be caused by the account's password expiring, which simply coincided with the LDAP integration work.

[14]Light-weight Authentication Protocol is used to centralize login policies in environments with many users and organizations.

3.2.2. Login and TCP/IP settings

In another case where the actual cause was due to password expiration, a customer wanted to add a certain storage subsystem to their enterprise storage *management* software. In other words, the customer desired to have a particular storage subsystem managed by an enterprise storage solution so that it would begin administering it automatically, along with their other servers and storage subsystems.

To this end, the management software offers a web-based GUI so that users can easily add subsystems they desire by specifying their IP addresses, after which a common handshake protocol is initiated in order to integrate the subsystems and begin managing them. However, this particular customer was using a brand new version of the management software which had different TCP/IP settings for the handshake than earlier versions. Hence, the first attempt at adding the subsystem failed. Immediately after modifying a TCP/IP parameter in the management software's console to allow the new subsystem's handshake, the management solution's web-based GUI became unresponsive and thereafter simply showed the HTTP message "*Error 500*". Further, reverting the



originally (Fig 7, before) *resulting*

Figure 10

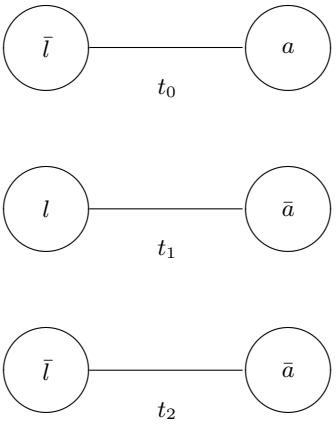


Figure 11: Progression of abstract states for example 3.2.1

value of the TCP/IP parameter did not alleviate the problem.

This situation can again be modeled with abstract states and reachability graphs as shown in figure 13, where A stands for "*result of authentication to web-based GUI*", and T represents "*value of TCP/IP parameter*". Note that t denotes the original value, while $\bar{t}$ denotes the modified value.

Just like in the previous example, reachability graphs and theorem 1 allowed us to deduce that variable T (the value of the TCP/IP parameter) was irrelevant to $\bar{a}$ (the inability to authenticate to the solution's web-based GUI). Given this information, we turned our investigation elsewhere and found that the web-based GUI was failing to connect to its database containing all the information for subsystems – information which it usually presented to the user right after login. It was this database's account whose password had expired, therefore causing the entire web-based GUI to issue the HTTP "Error 500" message. Once the account's password was changed, the customer was again able to authenticate to the web-based GUI and the TCP/IP parameter was modified, allowing the smooth integration of the new subsystem as desired.

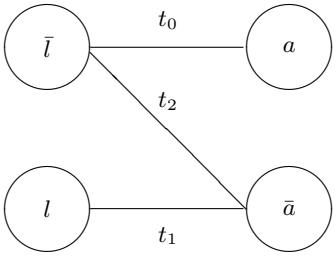


Figure 12: Reachability graph as the condensed state progression from figure 11.

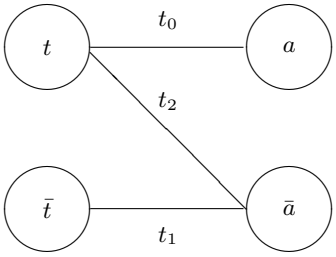


Figure 13: Condensed reachability graph for example 3.2.2

3.2.3. Email alerts not sent

Some data-center *monitoring* tools have features whereby alerts are sent to subscribers (such as data-center administrators) via email. Such alerts are useful not only for learning about exceptional situations that occur in customers' data-centers (such as "*primary site is down*"), but also for getting feedback about normal or expected events (such as "*network is operational again at the backup site*").

For a time, a particular customer received all desired alerts via email but then noticed one day that a subset of the alerts were no longer being sent. Additionally, the customer found that an important configuration file for the monitoring solution appeared to have the alert/email (SNMP[15]) functionality disabled, even though she had been receiving the alerts successfully before. The issue was further confounded by the fact that the customer had also applied a patch to fix a known bug in the monitoring solution's email functionality, albeit with no negative side-effects until the alert issue was detected.

This case has several potentially confounding situations, which can be modeled as follows. Let F represent the configuration file status with $\bar{f}$ denoting "*SNMP not explicitly enabled*". Further, let A represent the status of alerts, with $\bar{a}$ denoting "*All desired alerts not sent*," and let P represent the status of applying

[15] Simple Network Management Protocol.

the patch, with $\bar{p}$ indicating "*patch not applied*". Figure 14 illustrates the problem with all the known information and timings.

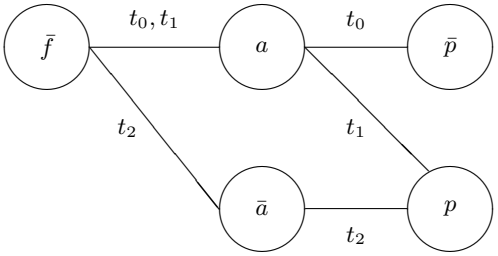


Figure 14: Condensed reachability graph for example 3.2.3

From the reachability graph in figure 14 and theorem 1, we can confirm that applying the patch is irrelevant to the issue with alerts. Further, we can also confirm that the problem is not related to SNMP not being explicitly enabled in the configuration file[16]. With potential confounders identified and removed thanks to theorem 1, we uncovered that an MIB[17] module had extra whitespace specifically in the lines related to the alerts in question, thus causing syntax errors. Once we fixed and verified the new MIB module[18], the issue was resolved and the customer began receiving the full set of alerts once again.

3.2.4. Database port change

This example shows how theorem 1 can illuminate *contributory* causes. It also shows how axiom 2 enables us to properly deal with information that reachability graphs provide at different times.

A customer needed to change their database's communication port used by one of their applications, but in doing so they found that the application no longer worked. The customer's application consisted of the back-end (the database) and middleware. The customer also decided to change the middleware's database port to match the new port utilized by the database, but when they did so, the application still did not work. This is modeled by figure 15.

Figure 15 emphasizes an aspect of theorem 1 as well as reachability graphs that readers might find confusing. In particular, notice that it would appear that A is irrelevant to M (i.e. that the value for the middleware's database port has no bearing on the application's working status). However, when controlling each variable separately so that the remaining known variables are held static at their original values, reachability graphs clearly demonstrate that M is relevant to A, as figure 16 shows.[19]

Hence, even though the reachability graph in figure 15 initially indicates that M is irrelevant to A[20] (i.e. $KB_{t_i} = \{M \not\sim A, D \sim A\}$), axiom 2 allows us to correct our knowledge of the SUT based on the actual relationship between A and M (shown by figure 16) to determine that it indeed influences the way the application works (i.e. $KB_{t_j} = \{M \sim A, D \sim A\}$, for $j > i$). The way we determined that $M \sim A$ was by changing the middleware's port value alone and thus *testing* the middleware in isolation[21].

Therefore, the knowledge imparted by the reachability graphs do not contradict each other (i.e. they do not say that the middleware is both relevant and irrelevant to the application). Rather, since we determined that the middleware was indeed relevant when using a constrained reachability graph (figure 16), we add that information to our knowledge base for the SUT and then use that knowledge with the extended reachability graphs to infer that properties D and M both *contribute* to the application's functionality. In other words, once we know which properties are relevant, our knowledge base is amended with that information as per axiom

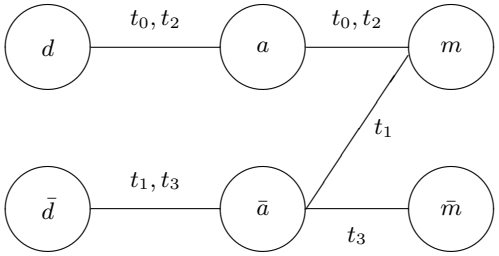


Figure 15: Condensed reachability graph for example 3.2.4. A represents "*application working status*" (a = "*application works fine*"), D represents "*database port value*" (d = "*DB port set to* α" and $\bar{d}$ = "*DB port set to* β") and M represents "*middleware's DB port value*" (m = "*middleware's DB port set to* α" and $\bar{m}$ = "*middleware's DB port set to* β"). Note that α and β represent specific port values.

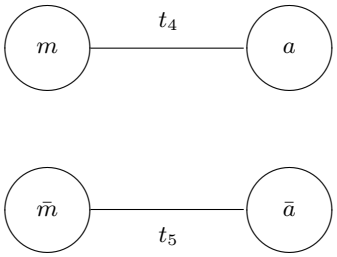


Figure 16: Reachability graph showing that M is relevant to A for example 3.2.4

[16] We later confirmed that by not explicitly enabling SNMP in the configuration file, the monitoring application utilized default ports, which the customer was using all along.

[17] Management Information Base

[18] http://www.simpleweb.org/ietf/mibs/validate/

[19] The relevancy of D on A is analogous to figure 16.

[20] Or, likewise, that D is irrelevant to A if we had changed M before D.

[21] And, likewise, doing the same with the database.

2 and such information does not change simply by extending the reachability graphs with more properties.

Going back to the customer case, it was clear that not all pieces of the puzzle were known since there were still other unknown components influencing the correct functionality of the application. With this information, further testing revealed that the application's front end (a web-based GUI) was also designed to connect directly to the database, so it also needed to change its database port. Once all three component ports were changed, the application then worked fine again. These contributory causes are illustrated using Pearl's graphical notation[22] in figure 17, where directed edges indicate that target nodes are affected by (i.e. influenced by) the associated source nodes. With the new information, we then have $KB_{t_k} = \{F \sim A, M \sim A, D \sim A\}$, for $k > j$.

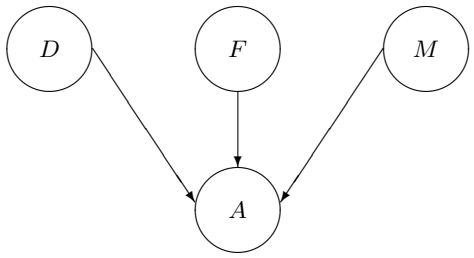


Figure 17: Causal diagram for example 3.2.4 showing that A is affected by D, F, and M. F represents "*front end port value*".

4. HOW EFFECTIVE FAULT LOCALIZATION WORKS, PART 2

Theorem 1 works due to the fact that simpler models[23] are usually sufficient to explain causal relationships as opposed to more complicated models. Such simpler models directly represent the use of the problem-solving principle called *Occam's razor*.

However, even though theorem 1 is a necessary criterion to identify and isolate irrelevant properties – which is itself an effective fault localization method – it is not *sufficient* when causal models are more complicated. In particular, theorem 1 is not sufficient in *preemptive* situations, whereby the value of a certain set of variables, V, cause other variables, $\hat{V}$, to become irrelevant even if those other variables in $\hat{V}$ might also be causes at other times and with different values [7].

To overcome this limitation, we utilize Pearl's *Test for Stable No-Confounding*, which is an extension of the Associational No-Confounding criterion (definition 1) but with the power of a key observation based on the concept of *stability* of models [7]. In short, stable causal structures do not have causal relationships created or destroyed simply by varying the conditions of the environment. In contrast, unstable causal models contain temporary causal relationships that might emerge simply because certain conditions are "just right"[24], thus masking the root cause (or contributory causes) in seemingly random ways. Such confoundedness is also called *incidental* confoundedness [7].

[22] Also called *causal diagrams*.

[23] Which Pearl calls *minimal causal structures* [7]

[24] As in the case of preemption and race conditions

Therefore, the following theorem is able to help us distinguish whether confoundedness is at play and, additionally, whether it is stable or unstable. The theorem is the second part of the effective fault localization methodology presented in this paper.

THEOREM 2. ***Test for Stable No-Confounding***

Let A_Z denote the assumptions that (i) the causal relationship data are generated by some (unspecified) acyclic causal model M and (ii) Z is a variable in M that is unaffected by X but may possibly affect Y.

If both the associational criteria in definition 1 (Associational No-Confounding) are violated, then X and Y are not stably unconfounded given A_z.

In other words, theorem 2 states that if any *one* of the conditions C_1 or C_2 of definition 1 are satisfied (given the requirements set forth by theorem 2), then X and Y are not confounded and, further, the non-confoundedness is stable. However, if both conditions are violated, then the causal relationships between X and Y are not stable, meaning that they might be incidentally (un)confounded simply by changing the conditions of the environment, thus removing or creating confoundedness only in certain situations but not others.

The power of theorem 2 stems from the fact that it "does not require us to know the causal structure of the variables in the domain or even to enumerate the set of relevant variables" as in definition 1, for which we had to have information about *all* the variables $Z \in T$ [7]. Further, to determine whether or not X and Y are stably non-confounded, we only need to find a *single* variable Z (satisfying A_Z) and use the conditions of definition 1 as the test [7]. Indeed, as Pearl explains: "the *qualitative* assumption that a variable *may* have influence on Y and is not affected by X suffices to produce a necessary statistical test for stable no-confounding"[25] [7].

Just as example 3.2.4 showed for theorem 1, theorem 2 can also indicate when further investigation is needed. For instance, when no irrelevancy can be determined by using theorem 1 with the information at hand, one would then proceed to determine whether the variables of interest are stably confounded. However, since theorem 2 can only be used with more than two variables that we think might be at play, further investigation is signaled to find that third variable Z in the problem space. That is, to determine whether or not two variables of interest are confounded – and whether that (un)confoundedness is stable – a third variable is needed that meets the requirements for theorem 2, thus prompting the search for such a variable.

Theorem 2, therefore, is useful as a second step in a robust fault localization methodology (after first applying theorem 1) for two important reasons. First, it offers a test for an important property of stability in causal models. Second, it is capable of indicating when further investigation is required elsewhere in the problem space.

4.1. Applications of Theorem 2

The following example illustrates the utilization of the fault localization methodology presented in this paper as a whole.

[25] Italics mine.

4.1.1. Build Broken by Tests

A customer's development team were in the process of adding new features to one of their flagship software products. Being test-driven, the development team ensured that each new piece of code was successfully tested by corresponding unit and integration tests. However, once they began to build the actual packaged application as part of their continuous integration and release cycles, they stumbled into a frustrating and money-loosing scenario whereby their continuous integration process did not complete because some of the tests it invoked were simply hanging while running on the packaged product[26]. To make matters more confusing, their test team found no such issues with the packaged application during manual testing.

Initially, it was believed that either the new code or the new tests were causing the hang. Naturally, reverting to the previous code version removed the hang, which allowed the build to complete. However, simply deactivating the new code[27] also removed the hang but not in all cases. Therefore, since deactivating the new code did not consistently remove the hang, this indicated that the new code was merely a contributory cause for the problem, and the actual cause was somewhere else. It also suggested that *incidental* confoundedness might be at play. To verify this hypothesis and to determine whether the confoundedness was stable, we used theorem 2 as follows.

Let Y denote the event when the build hangs. Also let X_{new_code} denote the case when the new code executes. Let $X_{deactivate}$ represent the event when the code has been deactivated. As per theorem 2, the two conditions for this case resolve to (here $Z = X_{deactivate}$ and $X = X_{new\_code}$):

$C_1 : P(X|Z) \neq P(X)$

$C_2 : P(Y|Z, X) \neq P(Y|X)$

In other words, C_1 fails to hold because deactivating the new code clearly has an effect on whether the new code executes (i.e. the events X and Z are not independent). C_2 fails to hold because deactivating the new code did not resolve the hang in all cases. Since X_{new_code} and Y were confirmed to *not* be stably unconfounded, the search turned to other variables influencing the cause-effect relationships. Recall that one of the central questions was "Does the new code cause the hang?" The result of theorem 2 allowed us to assert with certainty that it was not entirely the root cause, although it somehow exposed the hang.

In this manner, several iterative applications of the methodology presented in this paper allowed us to direct our search to other relevant variables and practically ignore others that we previously suspected. For instance, the methodology enabled us to completely rule out the new unit and integration tests as contributing to the hang. This was because randomly disabling and enabling tests – as guided by theorem 1 – sometimes produced the hang and sometimes did not, even after re-enabling tests that previously led to hangs. We utilized theorem 2 to confirm that the tests and the hang were not stably unconfounded, thus alleviating the concern about the new tests being the cause and prompting the search for other relevant variables. In other words, the tests were indeed "trustworthy" and telling us something valuable.

By continuing to use the methodology as described in this paper, the cause was finally localized to the continuous integration process itself. The fault was found in the code that captured the output from the tests' stdout and stderr streams. Then, by merely swapping two lines of code within the continuous integration process so that stdout was read before stderr, the hang disappeared completely. Moreover, theorem 2 showed that we had finally arrived a a stable non-confounded variable pair. In this case, it meant that we had effectively honed in on the root cause of the problem without confounders nor instability[28].

[26] This case caused many developers and testers at the customer site to express concerns very similar to those shared by Joe Harter (see the introduction to this paper.) Indeed, such unfounded initial reactions in this particular case prompted the writing of this paper.

[27] As guided by theorem 1.

5. CONCLUSION

This paper has mathematically explained the difficulties underlying fault localization methods in terms of probabilistic associations and confounding bias. It has then provided a methodology for effective fault localization in the form of a conceptual framework with several important axioms, definitions, theorems, and a conceptual device called reachability graphs. The framework and methodology both aim to increase effectiveness of fault localization during software testing and debugging. Several examples are provided to elucidate how to effectively use the methodology and the conceptual framework so as to minimize the impact of confounding bias in order to localize faults to a high degree when performing software testing of any kind.

6. REFERENCES

[1] P. Ammann, and J Offutt, Introduction to Software Testing, Cambridge University Press, 2008, ISBN 978-0-521-88038-1

[2] C. Kaner, A Course in Black Box Software Testing, 2004

[3] W. E. Wong, and V. Debroy, "A Survey of Software Fault Localization," 2009.

[4] M. Renieris, and S. P. Reiss, "Fault Localization With Nearest Neighbor Queries," 2003.

[5] M. A. Alipour, "Automated Fault Localization Techniques; A Survey," Oregon State University.

[6] J. Xuan, and M. Monperrus, "Test Case Purification for Improving Fault Localization," 2014.

[7] Judea Pearl. 2013. Causality: Models, Reasoning, and Inference. Cambridge University Press, New York, NY, USA.

[8] G. K. Baah, A. Podgurski, and M. J. Harrold, "Matching Test Cases for Effective Fault Localization," 2011.

[9] M. Papadakis, and Y. Le Traon, "Using Mutants to Locate Unknown Faults," 2012.

[28] The root cause of the hang: it turned out that the new code caused stdout to receive more data from the tests than its buffer could fit. The particular implementation of the stream-handling code utilized by the continuous integration process forced stderr's buffer to remain empty as long as stdout had incoming data to be read. In other words, stdout's buffer got filled first, before stderr's buffer. However, the continuous integration process was coded to read data from stderr first, implying three things: 1) no stdout data was being read and its buffer was not getting cleared, 2) not clearing stdout's buffer prevented data from being placed in stderr's buffer, 3) reading from stderr would block until data became available. These conditions caused the build to hang. Therefore, swapping the two lines of stream-reading code resolved this problem.

[10] J. Kim, J. Park, and E. Lee, "A New Spectrum-based Fault Localization with the Technique of Test Case optimization," 2013.

[11] W. E. Wong, "On The Equivalence Of Certain Fault Localization Techniques," 2011.

[12] A. Avizienis, J. Laprie, and B. Randell, "Fundamental Concepts of Dependability," 2000.

[13] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for Java," 2005.

[14] J. de Keer and B. C. Williams, "Diagnosing Multiple Faults." Artif. Intell., 32(1):97-130, 1987.

[15] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking." Software - Practice and Experience, 23(6):589-616, 1993.

[16] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "Program spectra analysis in embedded systems: A case study," 2006.

[17] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," 2005.

[18] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," IBM Systems Journal, 41(1):4-12, 2002.

[19] L. J. Morell, "A theory of fault-based testing," 1990.

[20] M. Renieris, and S. P. Reiss, "Fault localization with nearest neighbor queries," 2003.

[21] A. Zeller, "Isolating cause-effect chains from computer programs," 2002.

[22] G. K. Baah, A. Podgurski, M. J. Harrold, "The Probabilistic Program Dependence Graph and Its Application to Fault Diagnosis," 2008.

[23] A. Banerjee, A. Roychoudhury, J. A. Harlie, and Z. Liang, "Golden implementation driven software debugging," 2010.

[24] S. Bates, and S. Horwitz, "Incremental Program Testing using Program Dependence Graphs," 1993.

[25] P. Cousot, and R. Cousot, "Abstract Interpretation: A Unified Lattice Model For Static Analysis of Programs by Construction or Approximation of Fixpoints," 1977.

[26] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying Bug Signatures Using Discriminative Graph Mining," 2009.

[27] H. Cleve, and A. Zeller, "Locating Causes of Program Failures," In Proceedings of the International Symposium on the Foundations of Software Engineering, pg 342-351, 2005.

[28] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable Statistical Bug Isolation," In Proceedings of the Conference on Programming Language Design and Implementation, pg 15-26, 2005.

[29] U. Kanewala, and J. M. Bieman, "Techniques for Testing Scientific Programs without an Oracle", 2013.