Using Model-Based Diagnosis to Improve Software Testing

Roni Stern
roni.stern@gmail.com

Meir Kalech
kalech@bgu.ac.il

Niv Gafni
gafniv@gmail.com

Yair Ofir
yair87@gmail.com

Eliav Ben-Zaken
zkena2@gmail.com

Dept. of Information Systems Engineering,
Ben Gurion University of the Negev, Israel

Abstract

It is often regarded as best-practice that the case the developer that writes a program, and the tester that tests the program, are different people. This supposedly allows unbiased testing. This separation is also motivated by economic reasons. As a result, the tester, especially in black-box testing, is oblivious to the underlying code. Thus, when a bug is found, the tester can only file a bug report, which is later passed to the developer. The developer is then required to reproduce the bug found by the tester, diagnose the cause of this bug, and fix it. The first two task are often very time consuming.

In our research, we aim at improving the process of software testing, by automatically directing the tester to provide focused testing when a bug is found. The purpose of these additional tests is to gather information that will be used by an automatic software diagnosis algorithm that will identify the root cause of the bug. The programmer is then given, in addition to the traditional bug report, the exact software component that caused this bug. Then, the time of the programmer is only spent on fixing the bug in the faulty software component. As a result, this focused testing and diagnosis process can result in substantial savings of programmer time, at the expense of minimal additional effort by the tester.

1 Introduction

Testing is a fundamental part of the software development process [Myers et al., 2004]. Often, most of the testing is done by Quality Assurance (QA) professionals, and not by the actual programmers that wrote the tested software. This separation, between those who write the code and those who test it, is often regarded as a best-practice, allowing a more unbiased testing. Additionally, this separation is often motivated by economic reasons, as programmers are in general more expensive than QA professionals.

As a result of this separation, when a bug is found by the tester, it cannot be immediately fixed, as the tester may not be familiar with the tested code. This is especially true in *black box* testing (also known as *functional testing*), where the tester is completely unaware of the tested code. Therefore, the common protocol when a tester finds a bug is that this bug is reported in a bug tracking systems, such as *HP Quality Center* (formerly known as *Test Director*), *Bugzilla* or *IBM Rational ClearQuest*. Periodically, the reported bugs are prioritized by the product owner, and the programmers begin to repair the reported bugs.

Fixing such reported bugs usually involves two tasks. First, the programmer needs to diagnose the root cause of the bug. Then, the programmer attempts to repair it. Diagnosing the root cause of a software bug is often a challenging task that involves a trial-and-error process: several possible diagnoses are suggested by the programmer, which then performed tests and probes to differentiate the correct diagnosis. This trial-and-error process has several challenges. It is often non-trivial to reproduce bugs found by a tester (or an end-user). Also, reproducing a bug in a development environment may not represent the real (production or testing) environment where the bug has occurred. Thus, the diagnosis, and correspondingly the patch that will fix the bug in the development environment, may not solve the reported bug in the real environment.

Note that since the tester is not familiar with the tested software, he is obligated to a predefined test suite. Otherwise, the tester might have performed additional tests when a bug is observed to assist the programmer in finding the correct cause of the bug.

In our research, we aim at improving the software testing process described above, by combining diagnosis and planning algorithms for the field of Artificial Intelligence. Model-Based Diagnosis algorithms have been proposed in the past for the purpose of diagnosing software bugs [González-Sanchez et al., 2011; Abreu et al., 2011; Wotawa and Nica, 2011; Stumptner and Wotawa, 1996]. Thus, when a test fails and a bug is found, one can use these algorithms to generate automatically a set of candidate diagnoses.

To identify which of these candidate diagnoses is indeed the correct diagnoses, we propose several algorithms for suggesting additional, focused, testing steps for the tester. These tests are generated automatically, by considering the set of candidate diagnoses and proposing tests that will allow to differentiate between these candidate diagnoses, until a single diagnosis is found. In this paper we propose several algorithms for planning these additional focused testing. In particular, we propose to view the task of generating an effective focused test case as a "planning under uncertainty" problem, and solve it using a Markov Decision Process solver from the Artificial Intelligence literature.

The paper is structured as follows. First we describe in high-level the proposed approach to improve software testing. Then, we describe in detail the two main components of the proposed approach: the diagnosis algorithm and the test planning algorithm. Following, we describe preliminary experimental results. Finally, we discuss the long-term vision of the proposed approach, and briefly describe future work.

2 Artificial Intelligence in Software Testing

Consider the common black-box software testing process, depicted in the left side of Figure 1. For ease of notation we refer to testing engineer that performs the test (e.g., the QA professional) as simply the *tester*, and refer to the software developer as the *developer*.

2.1 Traditional Approach

The tester executes a test suite (a sequence of tests), until either the test suite is done and all the tested have passed, or one of the tests fails, in which case a bug has been found. As is often the case, the developer and the tester are different individuals, to allowing a supposedly more unbiased testing. Thus, the tester is not expected and often is unable to immediately fix the bug that was found. As mentioned in the introduction, this is especially true in *black box* testing (also known as *functional testing*), where the tester is completely unaware of the tested code. In such cases, the tester files a bug report in some bug tracking systems (e.g., *HP Quality Center*, *Bugzilla* or *IBM Rational ClearQuest*), and continues to test other components (if possible).

Periodically, the reported bugs are prioritized by the product owner, and passed on to the developer to fix them. Most commonly, the developer that the bug is assigned to will perform the following tasks to fix the bug.

1. Reproduce the bug in the developer environment, e.g., on the workstation of the developer.
2. Identify the root cause of the bug, i.e., the software component that is faulty.
3. Fix the faulty component.

Surprisingly, the first step of reproducing a bug can be a difficult and time consuming task. This is because bug reports may be missing important details, which are required to reproduce the bug. Furthermore, programs often have a state (e.g., executing a transaction) that affects the behavior of the program. Reproducing a given state of the program may also be non-trivial and time consuming. Additionally, some programs contain a stochastic element, which makes reproducing a bug even harder.

After the bug is reproduced, the developer identifies which component is believed to be faulty, and then fixes that component. Note that in order to identify which component is faulty, the developer often tests various parts of the system. The faulty component is then identified by observing the behavior of the system under these tests.

Naturally, this is an iterative process. After the developer fixes the bug (and performs initial tests to verify that the bug is fixed), the tester repeats the failed test suites[1], to verify that the bug has indeed been solved. This entire process is listed in the left side of Figure 1.

[1] It is also possible that the tester also performs the passed tests, to verify that the developer did not cause a new bug while fixing the other bug.

2.2 Proposed Approach

In this paper we propose to improve this traditional test-and-fix process by empowering the tester with tools from Artificial Intelligence (AI). In particular, we propose to use existing AI techniques to guide the tester to perform the additional tests that the developer would have performed, in order to identify which software components (e.g., function, class) are faulty. As a result, the developer is focused on the faulty component, and can invest more effort in fixing the bug. This proposed approach is listed on the right side of Figure 1.

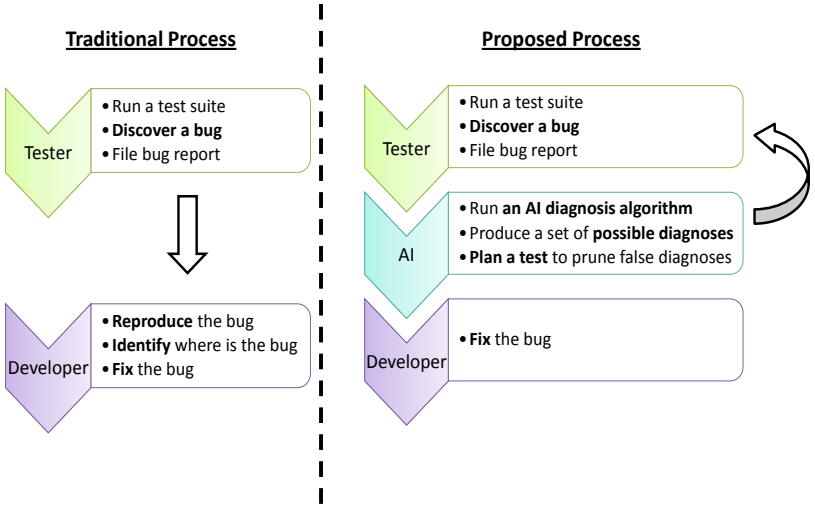

Figure 1: The traditional vs. the proposed approach for software testing.

The proposed approach is composed of two main components:

1. A *diagnosis algorithm*, that can infer from the software source code and the observed tests a set of possible diagnoses. Such diagnosis algorithms have been proposed in the recent years, especially for the purpose of diagnosing software faults ("bugs"). In the following section (Section 3) we briefly review one of these algorithms, which is easy to implement and can scale to large systems.

2. A *planning algorithm*, that suggests further tests for the tester, to narrow the set of possible diagnoses. Several such algorithms are proposed in Section 4. In particular, we identify and address the tradeoff between diagnosis accuracy and effort of the tester.

Next, we discuss the *diagnosis algorithm* component.

3 Model-Based Diagnosis for Software

The most basic entity of a diagnosis algorithm is the *component*. A component can be defined for any level of granularity of the diagnosed software: a class, a function, a block etc. The granularity level of the component is determined according to the granularity level that one would like to focus on. Naturally, low level granularity will result in a very focused diagnosis (e.g., pointing on the exact line of code that was faulty), but will require more effort in obtaining that diagnosis.

The task of a diagnosis engine is to produce a *diagnosis* which is a set of components that are believed to be faulty. In some cases, diagnosis algorithms return a set of *candidate diagnoses*, where each of these candidates can potentially be a diagnosis, according to the observed tests.

There are two main approaches that have been proposed in the model-based diagnosis literature for diagnosing software faults (i.e., bugs). The first approach [Wotawa and Nica, 2011] considers a system description, which models in logical terms the correct functionality of the software components. If an observed output deviates from the expected output as defined by the system description, then logical reasoning techniques are used to infer the possible faulty components (diagnoses) that explain the unexpected output. Although this approach is sound and complete its main drawback is that it does not scale well. Additionally, modeling the correct behavior of every system component is often infeasible. Therefore, we focus in this paper on the other approach for software diagnosis, which is described next.

An alternative approach to software diagnosis has been proposed by Abreu et. al. (2011). In this approach, which is called the *spectrum-based* approach, there is no need to model the correct functionality of each of the software components in the system.

The *specturm-based* approach only requires for every executed test the following:

- The *outcome* of the test, i.e, if the test has passed correctly or whether a bug was found. This is be done by the tester.
- The *trace* of a test. This is a log of the system components (e.g., functions, classes) that were used during this test. Such a trace can be obtained by using most common software profilers, such as Java's JVMTI, for example.

In case that the test has passed, we can assume that all the components in the log trace are healthy.[2] If a test failed, this means that at least one of the components in the log trace is faulty. This is equivalent to the term *conflict*, from the classical MBD literature Reiter [1987]; de Kleer and Williams [1987]. A conflict is a set of components, such that the assumption that they are healthy is not consistent with the model and the observation. Thus, a trace of a failed test is actually a conflict, since if all the components in the failed test were healthy, the test would not have failed.

Identifying conflicts is useful for generating diagnosis. This is because every diagnosis is a *hitting set* of all the conflict. A *hitting set* of a set of conflicts is a set of components that contains a representative component from each conflicts in the conflict

[2]Actually, there are diagnosis algorithms can also handle intermittent faults, where a faulty component may sometime output correct behavior. However, this is beyond the scope of this paper.

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	FAILED?
0	0	0	1	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0	0
0	0	1	0	0	1	0	1	1	0	1
0	1	0	0	0	0	0	0	0	1	1

Table 1: A set of 4 tests indicated by their success.

$\Delta_1 =$ $\{v_1, v_2\}$	$\Delta_2 =$ $\{v_1, v_5\}$	$\Delta_3 =$ $\{v_1, v_7\}$	$\Delta_4 =$ $\{v_1, v_8\}$	$\Delta_5 =$ $\{v_2, v_9\}$	$\Delta_6 =$ $\{v_5, v_9\}$	$\Delta_7 =$ $\{v_7, v_9\}$	$\Delta_8 =$ $\{v_8, v_9\}$
$1/6$	$1/6$	$1/6$	0	$1/6$	$1/6$	$1/6$	0

Table 2: Diagnoses: hitting sets of Λ_1 and Λ_2.

set Reiter [1987]. Intuitively, since at least one component in every conflict is faulty, a hitting set of the conflicts will explain the unexpected observation.

As an example of the spectrum-based approach and its relation to conflicts and diagnoses, consider a system with 10 components, $\{v_0, ..., v_9\}$. Figure 1 describes four tests performed on this system by a tester. The columns marked with '1' (except for the last column) represent the components have been invoked in the test (i.e., the components in the trace). The last column indicates whether the test passed (0) or failed (1), as reported by the tester. Hence, in this example the first two tests passed and the last two tests failed. Based on the failed tests we can generate the next conflicts: $\Lambda_1 = \{v_2, v_5, v_7, v_8\}$ and $\Lambda_2 = \{v_1, v_9\}$. The diagnoses that corresponds to the hitting sets of these conflicts are denoted by $\Delta_1, \ldots, \Delta_8$ and given in Table 2.

Naturally, given a trace of a single failed test with n components will result in n candidate diagnoses. This is because a diagnosis algorithm will not be able to determine which of the components in the trace has caused the bug. Performing more tests and as a result obtaining more conflicts and more traces of passed tests can modify the set of candidate diagnoses. This is because diagnoses must not contain components from traces of passed tests, and every diagnosis must also be a hitting set of all the conflicts - i.e., the traces of the failed tests. Furthermore, the *spectrum-based* approach [Abreu et al., 2011] also provides a mathematical formula to rank statistically the set of candidate diagnoses. The exact probability computations are given by Abreu et al. [2011]. In our example, shown in Table 2, the probability of the diagnoses are presented in the last row of Table 2. Notice that the probability of diagnoses Δ_4 and δ_8 is 0 since they involve component v_8 which was involved in a passed test. Thus, Δ_4 and Δ_8 are removed from the set of candidate diagnoses.

This *spectrum-based* approach to software diagnosis can scale well to large systems. However, it is not guaranteed to converge quickly to the correct diagnosis. Thus at the end of this process there can be a quite large set of alternative candidate diagnoses. Our purpose is to identify the correct diagnosis. Next, we describe how to automatically plan additional tests, to prune the set of candidate diagnosis, in an effort to find the correct diagnoses.

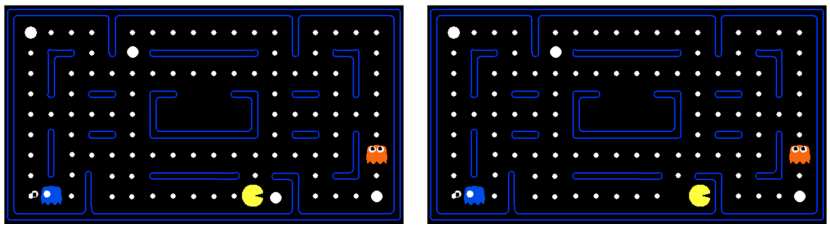

Figure 2: Pac-man example. (top) The initial pac-man position, (right) the position where the bug was observed

4 Automated Planning of Focused Software Testings

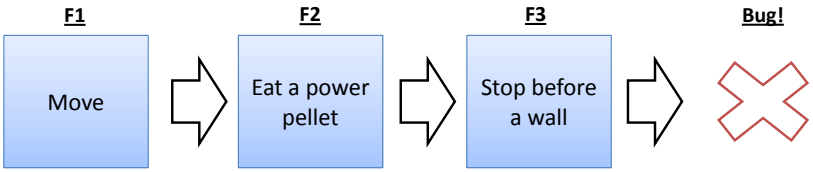

Figure 3: Simple high-level execution trace for the pac-man example.

The previous section reviewed a feasible algorithm for finding a set of candidate diagnoses, given a set of observed execution traces that contain a software bug. However, for a given observation, there is often many candidate diagnoses, but only one correct diagnosis. This is why the developer often performs additional tests[3] to identify which diagnosis is correct. In this section we propose a family of algorithms that plan a sequence of tests to narrow down the number of possible diagnoses. These tests will be generated by these algorithms on-the-fly, when the tester reports a bug. The tester will then execute these focused testing, and as a result, the number of possible diagnoses will decrease, and the developer will be given a small set (or even a single) diagnoses to consider.

To illustrate how automated planning can be used to intelligently direct testing efforts, consider the following example. Assume that the developed software that is tested is based on a variant of the well-known pac-man computer game, depicted in Figure 2. We chose such a simplistic example for clarity of presentation. The left part of Figure 2 shows the initial state of this variant of the pac-man game. Assume that the test performed by the tester is where pac-man moved one step to the right. The new location of pac-man has a power pellet (the large circle) and it is bordering a wall (Figure 2, right). Now, assume that following this test, the game crashed, i.e., a bug occurred. Also, assume that the trace of this test consists of three functions, as shown

[3]This can be, for example, running the program in debug mode and entering various parameters.

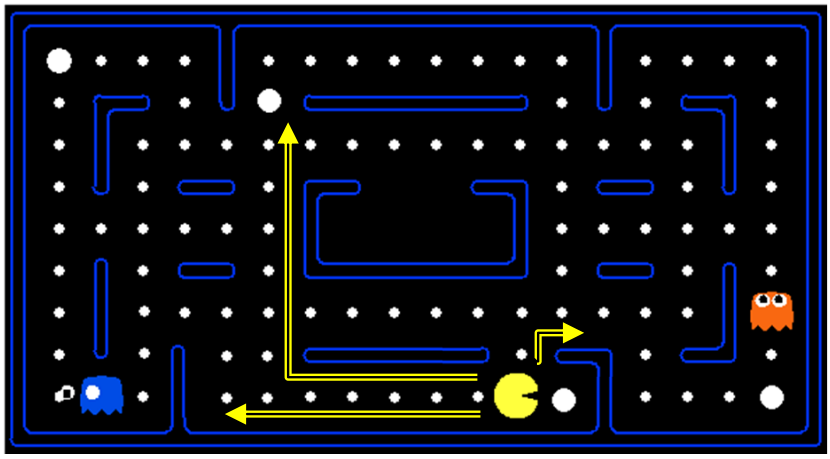

Figure 4: Pac-man example. The possible tests to perform.

in Figure 3: (1) Move right (denoted in Figure 3 as **F1**), (2) Eat power pellet (**F2**), and (3) Stop before a wall (**F3**).

There are at least three explanations to the observed bug: (1) the action "move right" (F1) failed, (2) the action "eat power pellet" failed, (3) touching the wall caused a failure.[4] It is easy to see that the diagnosis algorithm described in Section 3 would generate these three candidate diagnoses - $\{\{F1\}, \{F2\}, \{F3\}\}$. These candidate diagnoses can then be passed to the developer, to identify which of these functions caused the bug. After identifying the faulty function, the developer can fix it.

In this paper we go beyond this current testing process, where only the developer is required to identify the faulty function. In particular we will next propose automatic methods to plan additional focused test steps. Then, in addition to reporting a bug in a bug tracking system, the tester will also perform these focused tests. These tests will then be used by the diagnosis algorithm to identify the correct diagnosis. Then, the developer will only need to fix the diagnosed faulty component.

In our example, we can propose additional tests to check which of the three explanations is the correct one. By testing these explanations separately we can deduce the correct diagnosis. To check the correctness of the first explanation ("move right") the tester can move pac-man two steps up and one step to the right. To check the second explanation "eat power pellet" the tester should move pac-man to one of the other power pellets in the game. To check the third explanation, pac-man should be moved to the left wall. These three possible tests are shown in Figure 4, where each possible test is shown in a yellow arrow.

Generalizing the above example, the proposed testing process is given in Procedure 1 and described next. First, a set of candidate diagnoses is generated from the execution traces of the tests performed by the tester until the bug has occurred. This

[4]Naturally, the combination of these function could also cause the bug.

is done as described in Section 3. Then, a test case (i.e., a sequence of test actions for the tester) is generated, such that at least one of the candidate diagnoses is checked. To plan such a test, we consider the *call graph* of the tested software, which is defined next.

Definition 1 (call graph) *A call graph is represented by a directed AND/OR graph $G = (V, E_a, E_o)$, where V is a set of components and E_a and E_o are sets of edges. An edge between v_1 and v_2 represents a call from component v_1 to component v_2. E_o are 'or' edges representing conditional calls. E_a are 'and' edges representing regular calls.*

Planning a test that will check a given candidate diagnosis can be any executable path in the *call graph* that passes via a component that is part of that candidate diagnoses. Note that there are many automatic tools that generate a call graph from a static view of the source code.

Naturally, there can be many possible tests to check a given candidate diagnosis, and there may be many candidate diagnoses. In Section 4.1 we discuss intelligent methods to choose which test to perform. After the test is performed, the diagnosis algorithm is run again, now with the additional test that was performed. If a single diagnosis is found, it is passed to the developer, to fix the faulty software component. Otherwise, continue this process by planning and executing a new test. Naturally, one can define a timeout for this process, halting after a predefined amount of time and passing to the developer the (reduced) set of candidate diagnoses.

Algorithm 1: The focused testing process

Input: $Tests$, the tests performed by the tester until the bug was found.

1 $\Omega \leftarrow$ Compute diagnosis from $Tests$
2 **while** *Highest Ω contains more than a single diagnosis (or timeout has been reached)* **do**
3 $\quad NewTestPlan \leftarrow$ plan a new test to check at least one candidate diagnosis
4 $\quad$ Tester performs $NewTestPlan$, record output and trace in $NewTest$
$\quad Tests \leftarrow Tests \cup NewTest$ $\Omega \leftarrow$ Compute diagnosis from $Tests$
5 **end**
6 **return** Ω

Next, we describe several methods to plan these focused testing, such as to minimize testing effort required to find the correct diagnosis.

4.1 Balancing Testing Effort and Information Gain

Consider again the pac-man example, given in Figure 2. Recall that there are three proposed tests, marked by yellow arrows in Figure 4. Intuitively, one might choose to perform the first proposed test (move up twice and then right), since it demands the least number of steps. We assume for simplicity that the effort exerted by the tester when executing a test correlates with the number of steps in the test.

However, it is often the case that there are software components in which bugs occur more often. These components may be the more complex functions. For example, assume that in the pac-man example describe above, eating the power pellet is the most complex function (F2), which is more likely to contain a bug than the "move right" function (F1). These "bug-probabilities" can be given as input by the developer or system architect. There are even automatic methods that can learn these probabilities. For example, there are methods to predict which components are more likely to cause a bug, by applying data mining methods to project logs such as the source control history [Wu et al., 2011].

Given the probability of failure of every software components, we may prefer a test that checks the component with the highest probability, although it is expensive in terms of number of steps. Thus, in the pac-man example we might prefer to perform a test that checks if the function F2 (eating the power pellet) is faulty, instead of performing a test that checks if the function F1 (walking to the right) is faulty. The logic behind checking first the component that is most likely to be faulty is that it will reduce the overall testing effort of finding the correct diagnosis.

Alternatively, we may plan the next testing steps by considering both the fault probabilities as well as the testing effort (number of testing steps), in an effort to optimize a trade-off between the minimum testing steps with the highest fault identification probability.

Next, we describe several possible methods to plan and choose which test to perform. We use the term *focused testing methods* to refer to these methods. The overall aim of a *focused testing method* is to propose a sequence of test (one at a time) to minimize total testing effort (i.e. the number of test steps) required to find the correct diagnoses.

4.2 Myopic Focused Testing Methods

The first class of *focused testing methods* that we propose is called the *myopic* focused testing methods. The *myopic* focused testing methods contains two steps: (1) select a single component that we wish to check, and (2) return a test case that is the lowest cost (shortest) path in the call graph, passing from the entry point of the program via the selected component.

The *myopic* focused testing methods differ from one another by the way in which the checked component is selected. The simplest example of a *myopic* focused testing method is that which chooses the component that is closest to the entry point of the program. This corresponds to testing the component that is easiest to test, in terms of testing effort. We call this method the *lowest-cost* focused testing method.

Another example of a *myopic* focused testing method is the method which chooses to check the component that is most probable to be faulty. This can be calculated as follows by considering the set of candidate diagnoses and their probabilities. As described in Section 3, the software diagnosis algorithm of Abreu et. al. (2011) computes the probability of every candidate diagnosis, by considering prior probabilities of every software component being faulty, as well as the set of observed tests. Recall that a candidate diagnosis is a set of components. Thus, given a set of diagnoses and their probability, we compute the probability that a single component is faulty by taking the

sum over all the diagnoses that contain that component. For example, consider the set of diagnoses given in Table 2 and their probability. The component $v1$ is part of the diagnoses Δ_1, Δ_2, Δ_3, Δ_4, with probabilities $1/6, 1/6$, $1/6$ and zero. Thus, the probability that $v1$ is faulty is $3/6 = 1/2$. The *myopic* focused testing method that chooses the component with the highest probability is called the *highest-probability* focused testing method.

There are many possible hybrid *myopic* focused testing methods, which considers a combination of the lowest-cost and highest-probability. One possible approach is to consider a weighted sum of the probability and cost. Another approach is to choose the component that is closest to the entry point, but is faulty with probability higher than some predefined threshold.

4.3 MDP-Based Focused Testing Methods

A main drawback of the *myopic* focused testing methods is that they plan a test to check a single component at a time. Thus, they do not perform any long-term planning of the testing process. For example, it might be more efficient to consider a test that checks more than a single component. In addition, instead of performing a complete test case and then planning a new test according to the outcome of the entire test, it may be more efficient to replan a test case after every step in the test is performed.

More generally, we propose to view our problem as a problem of *planning under uncertainty* [Blythe, 1999]. Planning under uncertainty is a fundamental challenge in Artificial Intelligence, which is often addressed by modeling the problem as a Markov Decision Process (MDP). Once a problem is modeled as an MDP, it can be solved by applying one of wide range of algorithms such as Value Iteration, Policy Iteration [Russell and Norvig, 2010] and Real-Time Dynamic Programming [Barto et al., 1995]. A solution to an MDP is a policy, stating the best action to perform in each step. In our problem, this corresponds to returning the best test step the tester should perform, in terms of the expected future testing effort required until the correct diagnosis is found. Next, we describe how our problem can be modeled as an MDP. Once this is done, any MDP solver can be used.

An MDP consists of the following:

- a set of states, which describe the possible states that can be reached.
- an initial state, which is a state from which the process starts.
- a set of actions that describe the valid transition between states.
- a transition function, which gives the probability of reaching a state s' when performing action a in state s.
- a reward function, which gives the gain of performing an action in a given state.

Modeling our problem as an MDP can be done as follows. A state is the set of tests executed so far, the observed outcome of these tests. In addition, a state includes the current execution trace (of the current test). The initial state is the set of tests performed so far by tester, and an empty execution trace, which described the fact that the tester

is initially in the entry point of the program. The actions are the possible test steps that the tester can perform in a given state. The transition function should give the probability that a given test step will result in a bug or in a normal behavior. This is the probability that a component is faulty, where the component is the component that is reached in the given test step. Note that if a test step reaches several components, then the probability that this test step will encounter a bug is the probability that at least one of the components reached in this step causes a bug. The reward function in our problem is a negative reward: it is the cost of performing a single test step. An MDP algorithm seeks the policy that maximizes the expected reward that will be gained when executing that policy. Thus, in our case an MDP algorithm will seek the policy that minimizes the number of test steps until the correct diagnosis is found. This is exactly our goal - focus the testing effort, such that the correct diagnosis is found with minimal testing effort.

In conclusion, the MDP-based focused testing method works as follows. First, the problem is modeled as an MDP. Then an MDP solver is run to find an efficient policy. The tester is then given a single test-step, according the policy return by the MDP solver. After that single test-step is performed, the MDP solver is rerun, updating the initial state according to the outcome of this single test-step. Notice that one of the benefits of this approach is that a single test case can perform test steps that check more than a single test step.

It is not the goal if this paper to provide an empirical evaluation of the MDP-based and the *myopic* focused testing method. This paper aims at presenting a set of possible focused testing methods. However, we expect the MDP-based to be more efficient in terms of testing effort. On the other hand, the MDP-based method will also be the most computationally intensive, as the number of states in the proposed MDP can be very large, and the runtime of many MDP solvers is linear in the number of states in the MDP state space.

5 Preliminary Experimental Results

As explained above, we do not provide in this paper comprehensive experimental results to compare between the proposed focused testing methods. In this section we provide preliminary experimental results, only to demonstrate the applicability of the overall proposed approach. This is done as follows. A random graph is generated with 350 nodes, where every two nodes are connected by an edge with probability of 0.035. This graph represents the call graph of a diagnosed system. Every edge is an OR eedge or an AND edge with equal probability. Then, 2% of the nodes in the graph are chosen ranodmly to be faulty, and a set of 7 tests (observations) are also chosen randomly.

Following, we run the lowest-cost and the highest-probability myopic focused testing algorithms described in Section 4.2 on 25 difference scenarios, which were generated as described above. Figure 5 demonstrate the results of these experiments. The y-axis denotes the probability of the most probably candidate diagnoses. These probabilities are calculated by the spectrum-based diagnosis algorithm of Abreu et al. [2011], described in Section 3. Naturally, if this probability reaches one, we have the correct diagnosis. Thus, we refer to the values in the y-axis as a measure of the accuracy of

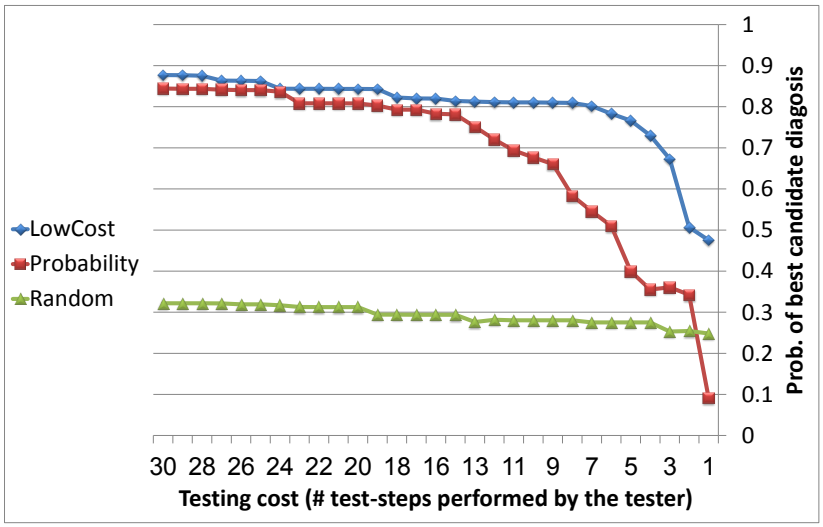

Figure 5: Preliminary experimental results on a 350 node call graph.

the current diagnosis candidate. The x-axis shows the cost, i.e., the number of testing steps required to achieve the probability given in the y-axis.

In these preliminary experiments we compared the performance of the myopic focused testing methods: lowest cost and highest probability. In addition, we ran as a baseline algorithm an algorithm that chooses the next test step randomly.

Several trends can be observed in the results shown in Figure 5. First, for each of the algorithms in the figure the y value increase with the value of x. This means that indeed, performing more tests results in a higher accuracy diagnosis. The second observation that can seen in Figure 5 is that the random base line is far worst than the more intelligent myopic focused testing method. While this is not surprising, it suggests that it will be worthwhile to develop intelligent algorithm for the focus testing method. Finally, we can also see that the lowest cost focused testing method outperforms the highest probability focused testing method. This suggests that ignoring the cost of a test and considering only its probabilities is not effective, and thus some combination of cost and probability seems in order. This is of course a topic for future work.

The above experimental results is very preliminary. We intend to extend the experimental results by also implementing the MDP-based algorithm described above. Also, we plan to perform extensive experiments on a range of graph sizes, and also seek to collaborate with a real software company, to perform a case study of the proposed approach.

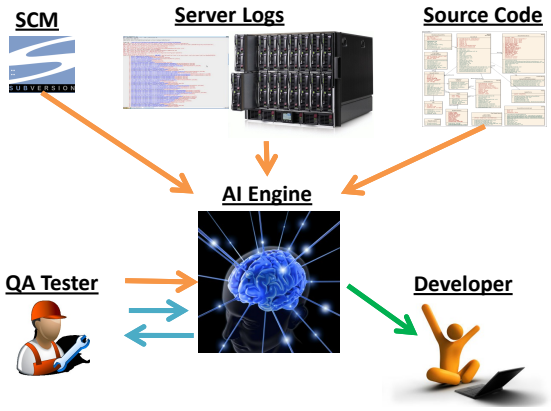

Figure 6: Long-term vision of incorporating AI diagnosis and planning into the testing process.

6 Discussion and Future Work

In the traditional testing process, when the tester finds a bug it files it in a bug tracking system. The developer is then required to identify which software component caused the bug, and fix it. In this paper we proposed a method that will identify for the developer the faulty software component that caused the bug, or at least provide a set of candidate faulty components. The proposed method is built from two components: (1) a diagnosis algorithm, that suggests a set of candidate diagnoses (i.e, a set of components that may be have caused the bug), and (2) a focused testing method, that will guide the tester to perform an additional set of tests, to identify which of the candidate diagnoses is the cause of the observed bug.

As a diagnosis algorithm, we propose to use the software diagnosis algorithm of Abreu et. al. 2011, which is a diagnosis algorithm that can scale to large systems and and do not require any modeling of the diagnosed software. The outcome of this algorithm is a set of candidate diagnoses, and the probability that each of these candidate diagnoses is correct.

The resulting set of candidate diagnoses may be large. We therefore propose several *focused testing methods* that are basically algorithms for planning new test actions for the tester to perform. The purpose of these additional tests is to select which of the candidate diagnoses generated by the diagnosis algorithm is indeed the correct diagnosis, i.e., contains the cause of the observed bug. Several such focused testing methods are proposed, in an effort to minimize the testing effort required to find the correct diagnosis.

In general, the aim of the proposed paradigm change proposed in this paper is to improve the software development process by using Artificial Intelligence (AI) tools to empower the tester and the testing process. This is part of our long-term vision of using AI techniques to improve the software development process, which is shown

in Figure 6. The AI engine will be given access to the source code, the logs of the software that are accumulated during the runtime of the software (e.g., server logs), and the source-control management tool (SCM) history. When a bug is detected, either in the software logs or by a (human or automated) tester, the AI engine will consider all these data sources, to infer the most probable cause of the bug. If needed, the tester will be prompted by the AI engine to perform additional tests, to help identifying the software component that caused the bug. This will be an interactive process, where the tester performs additional tests suggested by the AI engine, and reports the observed outcome of these tests back to the AI engine. Then, the developer will be given the faulty software component, and will be tasked to fix it. The developer can then report back when the bug was fixed, or to notify the AI engine that the bug was actually caused by a different software component. The AI engine will learn from this feedback to modify its diagnosis engine to avoid such errors in the future.

This paper presents only the first building block of this vision: automated diagnosis and automated focused testing methods. Future work on this building block will include empirical evaluation of the proposed focused testing method. In particular, this will be done first on synthetic call graphs and testing suites and random faults. Then, we intend to perform several case studies on real data, which will be gathered from the source control managements and bug tracking tools of a real software project in collaboration with existing software companies. We are now pursuing such collaboration.

Bibliography

Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Simultaneous debugging of software faults. *Journal of Systems and Software*, 84(4):573–586, 2011.

Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81 – 138, 1995.

Jim Blythe. An overview of planning under certainty. In *Artificial Intelligence Today*, pages 85–110. 1999.

Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1): 97–130, 1987.

Alberto González-Sanchez, Rui Abreu, Hans-Gerhard Groß, and Arjan J. C. van Gemund. An empirical study on the usage of testability information to fault localization in software. In *SAC*, pages 1398–1403, 2011.

G.J. Myers, T. Badgett, T.M. Thomas, and C. Sandler. *The Art of Software Testing*. Business Data Processing: a Wiley Series. John Wiley & Sons, 2004. ISBN 9780471469124.

Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

Markus Stumptner and Franz Wotawa. A model-based approach to software debugging. In *the Seventh International Workshop on Principles of Diagnosis (DX)*, pages 214–223, 1996.

Franz Wotawa and Mihai Nica. Program debugging using constraints – is it feasible? *Quality Software, International Conference on*, 0:236–243, 2011. ISSN 1550-6002. doi: http://doi.ieeecomputersociety.org/10.1109/QSIC.2011.39.

Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 15–25. ACM, 2011.