Using Model-Based Diagnosis to Improve Software Testing

Roni Stern [1], **Meir Kalech** [1], **Niv Gafni** [1], **Yair Ofir** [1], **Eliav Ben-Zaken** [1]

[1] *Department of Information Systems Engineering,*
Ben Gurion University, Beer-Sheva, Israel
roni.stern@gmail.com, kalech@bgu.ac.il,gafniv@gmail.com, yair87@gmail.com, zkena2@gmail.com

ABSTRACT

It is often regarded as best-practice that the programmer 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, where testers are often cheaper to hire than programmers. 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 suggest a combination of AI techniques to improve the above process. When a bug is found, a model-based diagnosis algorithm is used to propose a set of possible diagnoses. Then, a planning algorithm is used to suggest further test steps for the tester, that will help in identifying which diagnosis is the correct one. Then, the tester performs these tests and the diagnosis algorithm uses the knowledge acquired by these tests to identify the correct diagnosis or propose further tests. When the correct diagnosis is found, it is passed to the programmer, which then fixes it. This AI-driven process can save the programmer time substantially, at the expense of minimal additional effort by the tester. We propose and evaluate several planning techniques to minimize the extra work performed 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 programmers. This separation, between those who write the code and those who test it, is often regarded as a best-practice, supposedly allowing an 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 oblivious to 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.[1]

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;

[1] The exact title of the one which prioritizes the bugs depends on the structure of the software company. For example, in some cases this is the development team leader, in others case it is a representative of the clients.

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 planning additional, focused, testing steps for the tester. These tests are generated automatically, by considering the set of candidate diagnoses and proposing tests that will differentiate between these candidate diagnoses. This process of testing, diagnosing and planning further testing is repeated until a single diagnosis is found. In this paper we propose several algorithms for planning these additional focused testing.

The contributions of this paper is dual. First, we present a methodology change to the software testing and debugging process that uses model-based diagnosis and planning techniques (Section 2). Second, we propose several planning techniques for this problem that are meant to identify the correct diagnosis while minimizing the tests steps performed by the tester. These planning techniques are evaluated on simulated software model graphs.

2 ARTIFICIAL INTELLIGENCE IN SOFTWARE TESTING

In this section we propose our new paradigm for *software testing*. The purpose of this new paradigm is to improve the entire software development process by using model-based diagnosis and planning methods. Next, we explain the traditional *software testing* paradigm and supporting terms. Following, we present our new, AI-enhanced, testing paradigm.

2.1 Traditional Software Testing

The purpose of the *software testing* process (or simply *testing*) is to verify that the developed system functions properly. Testing is often performed by QA professionals, while the programming is done by programmers. We refer to the person that performs the testing as the *tester*, while the programmer will be referred to as the *developer*.

One can view *testing* as part of an information passing process between the tester and the developer. This process is depicted in the left side of Figure 1. The tester executes a predefined sequence of steps to test some functionality of the developed system. Such a sequence of steps is called a *test suite*. The tester runs all the steps in the test suite until either the test suite is done and all the tests have passed, or one of the tests fails. When a test fails - a bug has been found. The tester then 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).[2] Periodically, the reported bugs are passed to the developer, possibly prioritized by the product owner.

Most commonly, a developer that is given a bug to fix will perform the following tasks.

[2]According to some paradigms, the tester should try to continue the test suite even after a bug has been found.

1. **Diagnose.** Identify the root cause of the bug, i.e., the software component that is faulty.
2. **Fix.** Repair the faulty component.

To identify the faulty components, the developer often tests various parts of the system. The faulty components are then inferred by the developer by observing the behavior of the system under these tests. Once the faulty components are found, the developer fixes them.

2.2 The Test, Diagnose and Plan Paradigm

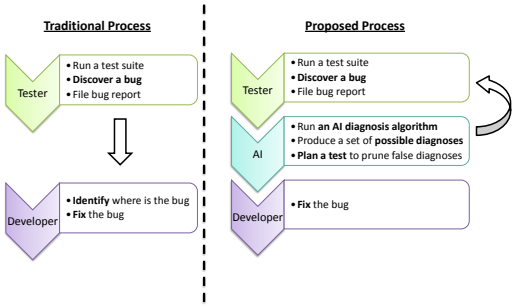

Figure 1: The traditional vs. the proposed approach for software testing.

In this paper we propose a new testing paradigm, which improve the traditional process described above by empowering the tester with tools from Artificial Intelligence (AI). In the proposed paradigm, when a bug is found a model-based diagnosis algorithm is run to suggest a set of candidate diagnoses. If this set contains a single diagnosis, then it is passed to the developer. Otherwise, a planning algorithm is used to suggest further testing steps for the tester that will narrow the set of possible diagnoses. The tester then performs these tests. The observed output of these new tests is given the diagnosis algorithm, which then outputs a new set of candidate diagnoses. This process is repeated until a single diagnosis is found and passed to the developer. Of course, other stopping conditions are also possible, and will be discussed later in the paper. We call this proposed paradigm the Test, Diagnose and Plan (TDP) paradigm. TDP is illustrated on the right side of Figure 1.

The great benefit of TDP over the traditional process described in the previous section is that in TDP the developer is given the exact software components that caused the bug. This information is given to the developer in addition to the traditional bug report. Thus, TDP trades the time of the tester for the time of the developer. This by itself is beneficial, as developers are often paid significantly higher salaries than testers.

However, we argue that the gain of using TDP is even greater, as follows. A fundamental part of finding what caused a bug is to reproduce the bug. Reproducing a bug in the development environment (e.g., the workstation of the developer) is often surprisingly difficult. This is because bug reports may be missing important details that are required to reproduce the bug. Also, programs may have a state (e.g., executing a

transaction, after the login page) that affects the behavior of the program. Thus, reproducing a bug may require getting to the same state that the tester that found the bug has been at. Another reason why reproducing a bug by the developer can be hard is because some programs contain a stochastic element. Hence, having the tester performs additional tests immediately when the bug is observed, as is done in TDP, is expected to be more efficient then having the developer reproduces the bug and search for its root cause. Even if the developer will still wish to reproduce the bug, it can be much easier to do so if one knows the components that caused it.

The TDP paradigm has two main components:

1. A **diagnosis algorithm**, that can infer from the observed tests a set of possible candidate diagnoses.
2. A **planning algorithm**, that suggests further tests for the tester, to narrow the set of possible diagnoses.

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.

Two main approaches have been proposed in the model-based diagnosis literature for diagnosing software faults (i.e., bugs). The first approach considers a system description that models in logical terms the correct functionality of the software components (Wotawa and Nica, 2011). If an observed output deviates from the expected output as defined by the system description, then logical reasoning techniques are used to infer candidate 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 in software.

An alternative approach to software diagnosis has been proposed by Abreu et. al. (2011; 2009), that is based on *spectrum-based fault localization (SFL)*. In this approach, there is no need to model the correct functionality of each of the software components in the system. All that is needed is the following information from every observed test:

- The **outcome** of the test, i.e, whether the test has passed correctly or a bug was found. This can be performed manually be done by the tester.

Trace										Outcome
v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	
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.

- The **trace** of a test. This is the sequence of the system components (e.g., functions, classes) that were used during this observed test. Such a trace can be obtained by using most common software profilers, such as Java's JVMTI, for example, or in an applicative system log, if implemented.

Components that are on the trace of a passed test are assumed to be healthy.[3] If a test failed, this means that at least one of the components in its trace is faulty. This corresponds to a *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 observation (and the system description). Identifying conflicts is useful for generating diagnosis. This is because every diagnosis is a *hitting set* of all the conflicts. A *hitting set* of a set of conflicts is a set of components that contains a representative component from each conflicts in the conflict 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 this SFL-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 trace of 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 candidate diagnoses that correspond to the hitting sets of these conflicts are therefore: $\{\{v_1, v_2\}, \{v_1, v_5\}, \{v_1, v_7\}, \{v_1, v_8\}, \{v_2, v_9\}, \{v_5, v_9\}, \{v_7, v_9\}, \{v_8, v_9\}\}$.

Performing more tests can narrow down the set of candidate diagnoses. This is because each of these candidate diagnoses must not contain components from traces of passed tests, and every candidate must also be a hitting set of all the conflicts - i.e., the traces of the failed tests. Thus, adding tests may decrease the number of candidate diagnoses.

An important aspect of the this SFL-based approach (Abreu *et al.*, 2011) is that it also provides a mathematical formula to rank the set of candidate diagnoses according to the probability that they are cor-

[3] Actually, some software diagnosis algorithms can also handle intermittent faults, where a faulty component may sometime output correct behavior. For simplicity, we assume in this paper that a faulty component will behave abnormally.

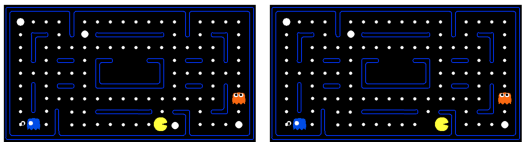

Figure 2: Pac-man example. (top) The initial pac-man position, (right) the position where the bug was observed

rect. The exact probability computations are detailed by Abreu et. al. (2011).

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.

4 PLANNING IN TDP

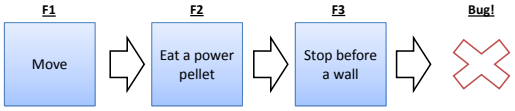

Figure 3: Simple high-level execution trace for the pac-man example.

The previous section reviewed a feasible algorithm for finding a set of candidate diagnoses, given a set of observed tests. Since there is often many candidate diagnoses, but only one correct diagnosis, further tests should be performed. 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 smaller set diagnoses (or even a single diagnosis) to consider. Importantly, our aim is to minimize the tester effort to find a single diagnosis.

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

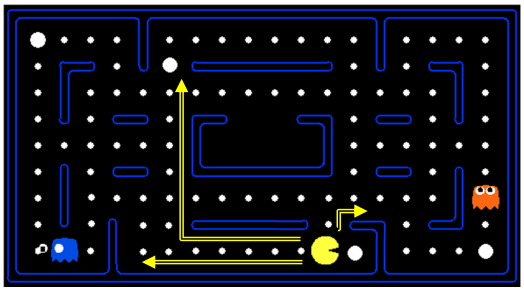

Figure 4: Pac-man example. The possible tests to perform.

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 - $\{\{F1\},\{F2\},\{F3\}\}$.

Further testing can be performed to deduce which of these three candidate diagnoses is the correct one. To check the correctness of the first candidate diagnosis (F1 - "move right") the tester can move pac-man two steps up and one step to the right. To check the second candidate (F2-"eat power pellet"), the tester should move pac-man to one of the other power pellets in the game. To check the third candidate diagnosis (F3-"touch a wall"), 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. By performing these additional tests, it is possible to deduce a single correct diagnosis.

Algorithm 1: An Algorithmic View of TDP

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 $NewTest$
5 $\quad Tests \leftarrow Tests \cup NewTest$
6 $\quad \Omega \leftarrow$ Compute diagnosis from $Tests$
7 **end**
8 **return** Ω

Generalizing the above example, Procedure 1 provides a more algorithmic view of TDP and how the test planning and diagnosis algorithms are integrated. First, a set of candidate diagnoses is generated from the observed tests that were performed by the tester until the bug has occurred (line 1 in Procedure 1). This is done as described in Section 3. Then, a test suite

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

(i.e., a sequence of test actions for the tester) is generated, such that at least one of the candidate diagnoses is checked (line 3). The tester then performs this newly generated test (line 4). After the test is performed, the diagnosis algorithm is run again, now with the additional information observed when the new test was performed (line 6). If a single diagnosis is found, it is passed to the developer, to fix the faulty software component. Otherwise, this process continues by planning and executing a new test.

Naturally, different stopping conditions can be defined for Procedure 1. 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. Another possible stopping condition is based on the probability that every candidate diagnosis is given by the diagnosis algorithm. When the set of candidate diagnoses contains a candidate diagnosis with probability that is higher than a predefined value, the process is halted and that diagnosis is passed to the developer. In our experiments, reported in Section 5, we have used a combination of these conditions, where if one of these conditions is met the experiment is halted. See Section 5 for details.

Next, we discuss how to plan new tests automatically. To do so, we first define what a *call graph* is.

Definition 1 (call graph) *A call graph is a directed AND/OR graph $G = (V, E_a, E_o)$, where V is a set of software components and E_a and E_o are a mutually disjoint 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.*

There are many automatic tools that generate a call graph from a static view of the source code. A test suite 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. As every step in the graph corresponds to an action of the tester, every test suite has a cost which is the cost (e.g., length) of its path in the graph.

Naturally, there can be many possible tests to check a given candidate diagnosis, and there may be many candidate diagnoses. Next, we describe several methods to plan these additional test suites, such as to minimize cost, which corresponds to the 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 test suites (one at a time) to minimize total testing cost required to find the correct diagnoses.

Probability-Based Focused Testing Methods
The first class of *focused testing methods* that we propose is based on the probabilities obtained by the diagnosis algorithm. Recall that the diagnosis algorithm assigns every diagnosis with a probability score, that marks how probable that diagnosis is to be correct. Using this probabilities, we propose two focused testing methods. The first, called *best diagnosis* (BD), first finds the most probable candidate diagnosis according to the probabilities mentioned above. Then, it plans the lowest cost path in the call graph that reaches at least one of the components in that candidate diagnosis. Another probability based focused testing method, called *highest probability* (HP), computes for every component the probability that it is faulty, by taking the sum over the probabilities of all the diagnoses that contain that component. HP then returns the lowest cost path that passes via the highest probability component.

Both of these probability-based approaches are motivated by the assumption that checking first high probability components will result in finding the correct diagnosis faster.

Lowest Cost Focused Testing Method
The next focused testing method that we propose is very simple. Plan the lowest-cost path that passes via at least one of the components in the set of candidate diagnoses, whose probability is not one or zero. We call this method the *lowest-cost* (LC) method. The last conditions of LC are important. If a component has a probability one for being faulty, there is no point in checking it, as every such test will fail. Alternatively, if a component has a probability of zero of being faulty, running a test through it will not reveal more knowledge about other faulty components.

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

Entropy-Based Focus Testing

All the previous focused testing methods choose a single component and then plan the shortest path to it. Thus, they completely ignore the components that are passed in the planned test except for the single chosen components. In the next focus testing method we propose, called the *high-entropy* (HE) method, we remedy this. The lowest-cost path to every component that exists in a candidate diagnosis (and does not have a probability one or zero) is considered. Then we compute the information gain of performing every such path, by calculating the entropy of the test suite that follows that path.

Calculating the entropy of a test suite T is done as follows. Let Ω_+ and Ω_- be the set of candidate diagnoses according to which T will pass and fail, respectively. Correspondingly, let $P(\Omega_+)$ and $P(\Omega_-)$ be the sum of the probabilities of each of the candidate diagnosis in Ω_+ and Ω_-, respectively. Then, the entropy of T is calculated as

$$P(\Omega_+)log(P(\Omega_+)) + P(\Omega_-)log(P(\Omega_-))$$

Note that it is theoretically possible to compute every possible path in the graph, measure the information gained by it and choose that path. However, the number of possible path in a graph is exponential, and thus the alternative entropy-based approach described above is preferred.

MDP-Based Focused Testing

All the previous methods, including the entropy-based, are *myopic*. They are *myopic* in the sense 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. 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).

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 and the observed outcome of these tests. The initial state is the set of tests performed so far by tester. The actions are the possible test suites that the tester can perform in a given state. The transition function should give the probability that a given test suite will fail or pass. This can be computed by the failure probabilities of the components in the test suite. As explained above, these probabilities are given by the diagnosis algorithms.

Before describing the reward function, we observe that since every state consists of a set of observed tests, one can run the diagnosis algorithm on them, and obtain a set of candidate diagnoses, each with a probability assigned to it. We call these set of candidate diagnoses the candidate set of the state. A state with a candidate set that contains a single diagnosis is regarded as a *terminal state*, as there is no point in making additional actions from it. Thus, our MDP can be viewed as a shortest path MDP, where we seek lowest-cost paths to a terminal state. Hence, we set the reward of a test suite is the negative value of its cost.

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.

Theoretically, solving the above MDP will yield the optimal policy, and hence will be the optimal focused testing method. However, the number of actions and states in this MDP is too large to solve optimally, since most MPD solvers are at least linear in the number of states in the MDP state space. It is easy to see that the number of states in our MDP is exponentially large (every possible set of tests and their outcome). We therefore perform the following relaxations. First, instead of considering all the possible test suites as possible actions, we use only a shortest path to every relevant (i.e., part of a candidate diagnosis) component as a possible action. This reduces the number of actions to be equal to the number of relevant components. Additionally, we set a probability threshold t that was used as follows. A state is regarded as a terminal state if its candidate set contains a candidate that has a probability higher than t. Thus, there is no need in the modified MDP to reach a state with a candidate set that contains a single candidate.

The last relaxation we used for our MDP is to bound its horizon by a parameter h (this is basically a bounded lookahead). This means that a state that is h steps from the initial state is also regarded as a terminal state. The reward of such states were modified to reflect how far they are from reaching a state that has a candidate diagnosis that has a probability t, as follows. For a state s let $p(s)$ be the probability of the candidate diagnosis in s that has the highest probability. Let s_{init} be the initial state, and let s' be a state on the horizon (i.e., at depth h). If $p(s') \geq t$ then its reward is zero (as no further testing are needed). Otherwise, the reward of s' is given

by $-(t-p(s'))\times\frac{h}{p(s')-p(s_{init})}$, which simply assumes that reducing the $p(s')$ value until it reaches t will continue in the same rate as it took to reduce $p(s_{init})$ to $p(s')$. For example, assume that $p(s_{init}) = 0.2$, $p(s') = 0.7$ and $h = 5$. This means that on average, every increase of h (i.e., every additional test) reduced $p(s_{init})$ by $\frac{0.7-0.2}{5} = 0.1$. Thus, we calculate the reward as $-(t - p(s')) \times \frac{h}{p(s_{init})-p(s')} = -(0.9 - 0.7) \times \frac{5}{0.7-0.2}$=-2. There are of course other options to calculate this reward.

There are many MPD solvers that can be used. In the experiments described below we set h to be three. Also, to save runtime we used a simple Monte-Carlo based MDP solver that samples the MDP state space to estimate the expected utility of every action.

Next, we compare experimentally the focused testing methods described above.

5 PRELIMINARY EXPERIMENTAL RESULTS

This paper does not presume to provide comprehensive experimental results to compare between the proposed focused testing methods. Furthermore, much more combinations and algorithms can be developed. However, we report in this section the results of a preliminary set experiments, in which we compared the proposed focused testing methods.

Every experiment was performed as follows. A random directed acyclic graph was generated with 300 nodes, where every two nodes are connected by an edge with probability of 1.3%. For every node in the graph a number of edges were set to be AND edges, while the other edges were set to be OR edges (see Definition 1). The number of AND edges was set as a random number in the range [1...6]. This AND/OR graph represents the call graph of a diagnosed system. Then, 2% of the nodes in the graph are chosen randomly to be faulty, and a set of 15 tests (observations) are also chosen randomly, to be the initial set of test performed by the tester before the bug was found.

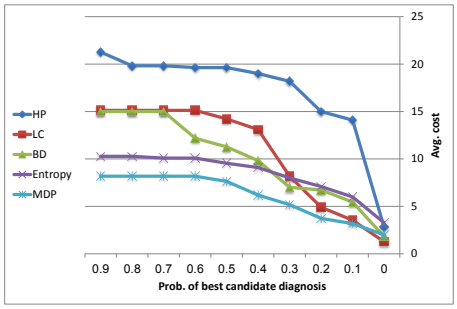

Figure 5: Preliminary experimental results on a 300 node call graph.

Following, we run all the algorithms described in Section 4.1 as part of TDP (Procedure 1. An experiment was halted when one of the following conditions were met: 1) a candidate diagnosis with probability higher than 0.9 was found, 2) a total of 160 TDP iterations were performed, and 3) if no new tests exists. The second condition was added to manage the total runtime of the experiments. The third condition exists, because there are cases where it is not possible to identify a single candidate diagnoses. This is known in the MBD literature as an ambiguity set. In such a case, no further testing will help, and the set of candidate diagnoses will be passed to the developer.

We performed a set of 20 such experiments, and measured the cost (the number of tested performed by the tester) required to find a candidate diagnosis with probability X, for $X = 0.1, 0.2, .., .0.9$. In 12 experiments from this set all the algorithms eventually found a candidate diagnosis with probability higher than 0.9. Figure 5 shows the results for this set of 12 experiments. The y-axis shows the average cost required to find a candidate diagnosis with probability equivalent to the x-axis value. Recall that the lowest-cost (Section 4.1), best-diagnosis and highest probability (Section 4.1) are denoted as LC, BD and HP, respectively. The entropy-based (Section 4.1) and MDP-based (Section 4.1) focused testing methods are named *Entropy* and *MDP* in the figure.

Results show that HP, LC and BD outperformed by the more sophisticated Entropy and MDP methods. MDP also outperformed all other methods substantially. This is reasonable, as the MDP method is the only method that is not myopic, in the sense that it plans for a sequence of test suites and not just the next test suite. The MDP method was also the most computationally intensive. The above experimental results are very preliminary. We leave to future work runtime comparison and a more comprehensive experimental evaluation, which will include analysis of the experiment parameters (e.g., graph size) and their effect on the performance of the different algorithms.

6 DISCUSSION AND FUTURE WORK

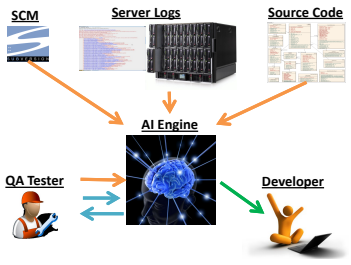

Figure 6: Long-term vision of incorporating AI diagnosis and planning into the testing process.

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 testing paradigm, called Test, Diagnose and Plan (TDP), where the tester, enhanced with AI techniques, will identify for the developer the faulty software component that caused the bug. TDP is built from two components: (1) a diagnosis algorithm, that

suggests a set of candidate diagnoses, and (2) a planning algorithm, 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 does 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. To reduce this set of candidate diagnosis and find the correct diagnosis, we propose to run a *focused testing methods*, which are algorithms for planning new test suites for the tester to perform. Several such methods were proposed, where the goal is 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.

REFERENCES

(Abreu *et al.*, 2009) Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 88–99, Washington, DC, USA, 2009. IEEE Computer Society.

(Abreu *et al.*, 2011) 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.

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

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

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

(González-Sanchez *et al.*, 2011) 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.

(Myers *et al.*, 2004) 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.

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

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

(Stumptner and Wotawa, 1996) 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.

(Wotawa and Nica, 2011) Franz Wotawa and Mihai Nica. Program debugging using constraints – is it feasible? *Quality Software, International Conference on*, 0:236–243, 2011.

(Wu *et al.*, 2011) 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.