Using Model-Based Diagnosis to Improve Software Testing

Tom Zamir and **Roni Stern** and **Meir Kalech**
tom.zamir.i@gmail.com roni.stern@gmail.com kalech@bgu.ac.il
Department of Information Systems Engineering
Ben Gurion University of the Negev
Be'er Sheva, Israel

Abstract

We propose a combination of AI techniques to improve software testing. When a test fails, a model-based diagnosis (MBD) algorithm is used to propose a set of possible explanations. We call these explanations diagnoses. Then, a planning algorithm is used to suggest further tests to identify the correct diagnosis. A tester preforms these tests and reports their outcome back to the MBD algorithm, which uses this information to prune incorrect diagnoses. This iterative process continues until the correct diagnosis is returned. We call this testing paradigm Test, Diagnose and Plan (TDP). Several test planning algorithms are proposed to minimize the number of TDP iterations, and consequently the number of tests required until the correct diagnosis is found. Experimental results show the benefits of using an MDP-based planning algorithms over greedy test planning in three benchmarks.

Introduction

Testing is a fundamental part of the software development process (Myers et al. 2004). A software testing phase involves finding bugs and fixing them. From the perspective of the programmer, fixing bugs usually involves two tasks. First, the root cause of the bug needs to be found, and then the faulty software components (e.g., functions or classes) are fixed. 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 performs tests and probes to differentiate the correct diagnosis. One of the reasons why this trial-and-error process is challenging is because it is often non-trivial to reproduce bugs found by a tester.

An ideal solution to this problem would be that the tester, when observing a bug, will perform additional test steps to help the programmer find the software component that caused the bug. However, planning these additional test steps cannot be done efficiently without being familiar with the code of the tested software. Often, testing is done by Quality Assurance (QA) professionals, which are not familiar with the software code that they are testing. This separation, between those who write the code and those who test it, is even regarded as a best-practice, allowing unbiased testing.

In this work we propose to enhance the software testing and debugging process described above by combining Model-Based Diagnosis (MBD) and planning techniques from the Artificial Intelligence (AI) literature. MBD algorithms have been proposed in the past for the purpose of diagnosing software bugs (González-Sanchez et al. 2011; Abreu, Zoeteweij, and van Gemund 2011; Wotawa and Nica 2011; Stumptner and Wotawa 1996). Thus, when a tester encounters a bug, any of these algorithms can be used to generate a set of possible diagnoses automatically.

To identify which of these diagnoses is correct, additional tests need to be performed. We propose several algorithms for planning these additional tests. These tests may be generated automatically or selected from a manually created set of tests, considering the set of possible diagnoses and choosing tests that will differentiate between them. This process of testing, diagnosing and planning further testing is repeated until a single diagnosis is found. Importantly, unlike previous work on test generation for software, we do not assume any model of the diagnosed software (Esser and Struss 2007) or an ability to manipulate internal variables of the software (Zeller 2002). One could consider this work as variant of algorithmic debugging (Silva 2011) that combines software MBD with planning.

The contributions of this paper are threefold. First, we present a methodology change to the software testing and debugging process that uses a combination of MBD and planning techniques. Second, we propose several test planning algorithms for identifying the correct diagnosis while minimizing the tests performed by the tester. Third, we evaluate these test planning algorithms on three benchmarks.

Background

We use the terms *tester* and *developer* to refer to the person that tests and programs the software, respectively. The purpose of traditional software testing process (referred to hereafter as simply *testing*) is to verify that the developed system functions properly. One can view *testing* as part of an information passing process between the tester and the developer, depicted in the left side of Figure 1. The tester executes a sequence of tests to test some functionality of the developed system. Such a sequence of tests is called a *test suite*. The tester runs all the tests in a test suite until either the test suite is done and all the tests have passed, or one of the tests failed. A failing test indicates the existence of a bug. To fix it, the tester passes information about the failed test to the developer, often in the form of a "bug report", and continues to execute other tests. The developer is then responsible for fixing the bugs found by the tester. This process is often performed using bug or issue tracking tools (e.g., HP Quality Center, Bugzilla, and IBM Rational ClearQuest).

In order to fix the bug, the developer needs to identify the faulty software component and then fix it. This process, of finding the faulty software component and fixing it, is commonly referred to as "debugging".

The Test, Diagnose and Plan Paradigm

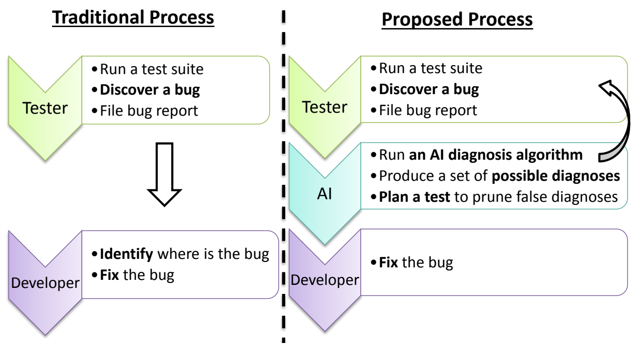

Figure 1: Traditional software testing vs. TDP.

We propose a new testing paradigm, called Test, Diagnose and Plan (TDP), for improving the testing and debugging processes described above by empowering the tester with tools from the Artificial Intelligence (AI) literature. TDP is illustrated on the right side of Figure 1. When a test fails, an MBD algorithm is run to suggest a set of possible *diagnoses*, i.e., software components that may contain a bug that caused the test to fail. If this set of diagnoses contains a single diagnosis, it is passed to the developer. Otherwise, a planning algorithm is used to plan further tests for the tester intended to narrow the set of diagnoses. The tester performs these tests and reports the observed output back to the MBD algorithm, which then outputs a new, potentially more refined, set of diagnoses. This process is repeated until a single diagnosis is found and passed to the developer. Other stopping conditions are also possible and discussed later in the paper.

The great benefit of TDP over the traditional testing and debugging is that in TDP the developer is given the exact set of software components that caused the bug. Moreover, having the tester perform additional tests immediately when the bug is observed, as is done in TDP, is expected to provide information which may be difficult to obtain when the bug is reproduced by the developer. This is because bug reports may miss important details and the tested software may have stochastic elements.

Algorithm 1 provides an algorithmic view of TDP and how diagnosis and test planning are integrated in it. It is called immediately after a test has failed. First, a set of diagnoses is generated from the tests performed so far by the tester (line 1). Then, an additional test is proposed, such that at least one of the diagnoses is checked (line 3). The tester then performs this test (line 4). After the test is performed, the diagnosis algorithm is run again, now with the additional information gained from the new test that 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 new tests.

Algorithm 1: An Algorithmic View of TDP

```
Input: $ObsTests$, the tests performed by the tester
       until the bug was found.
1 $\Omega \leftarrow$ Compute diagnosis from $ObsTests$
2 while $|\Omega| > 1$ do
3     $NewTest \leftarrow$ plan a new test
4     $NewObs \leftarrow$ Tester performs $NewTest$
5     $ObsTests \leftarrow ObsTests \cup NewObs$
6     $\Omega \leftarrow$ Compute diagnosis from $ObsTests$
7 end
8 return $\Omega$
```

The key components in TDP are the *diagnosis algorithm* used to compute diagnoses and the *planning algorithm* used to plan tests. We describe next how these components can be implemented.

Model-Based Diagnosis for Software

The input to classical MBD algorithms is a tuple $\langle SD, COMPS, OBS \rangle$, where SD is a formal description of the diagnosed system's behavior, $COMPS$ is the set of components in the system that may be faulty, and OBS is a set of observations. A diagnosis problem arises when SD and OBS are inconsistent with the assumption that all the components in $COMPS$ are healthy. The output of an MBD algorithm is a set of *diagnoses*.

Definition 1 (Diagnosis). *A set of components* $\Delta \subseteq COMPS$ *is a* diagnosis *if by assuming that they are faulty, then* SD *is consistent with* OBS.

In software, the set of components $COMPS$ can be defined for any level of granularity: a class, a function, a block etc. Low level granularity will result in a very focused diagnosis (e.g., pointing on the exact line of code that was faulty), but obtaining that diagnosis will require more effort. Observations (OBS) in software diagnosis are observed executions of tests. Every observed test t is labeled as "passed" or "failed", denoted by $passed(t)$ and $failed(t)$, respectively. This labeling is done manually by the tester or automatically in case of automated tests (e.g., failed assertions).

There are two main approaches for applying MBD to software diagnosis, each defining SD somewhat differently. The first approach requires SD to be a logical model of the correct functionality of every software component (Wotawa and Nica 2011). This approach allows using logical reasoning techniques to infer diagnoses. The main drawbacks of this approach is that it does not scale well and modeling the behavior of software component is often infeasible.

An alternative approach to software diagnosis has been proposed by Abreu et. al. (2011; 2009), based on *spectrum-based fault localization (SFL)*. In this SFL-based approach, there is no need for a logical model of the correct functionality of every software component in the system. Instead, the *traces* of the observed tests are considered.

Definition 2 (Trace). *A* trace *of a test t, denoted by $trace(t)$, is the sequence of components involved in executing t.*

Traces of tests can be collected in practice with common software profilers (e.g., Java's JVMTI). Recent work showed how test traces can be collected with low overhead (Perez, Abreu, and Riboira 2014). Also, many implemented applicative system log contain some form of this information.

In the SFL-based approach, SD is implicitly defined in SFL by the assumption that a test will pass if all the components in its trace are not faulty. Let $h(C)$ denote the health predicate for a component C, i.e., $h(C)$ is true if C is not faulty. Then we can formally define SD in the SFL-based approach with the following set of Horn clauses:

$$\forall test \;\; (\bigwedge_{C \in trace(test)} h(C)) \rightarrow passed(test)$$

Thus, if a test failed at least one of the components in its trace is faulty. In fact, a trace of a failed test is a *conflict*.

Definition 3 (Conflict). *A set of components $\Gamma \subseteq COMPS$ is a conflict if $\bigwedge_{C \in \Gamma} h(C) \wedge SD \wedge OBS$ is inconsistent.*

Many MBD algorithms use conflicts to direct the search towards diagnoses, exploiting the fact that a diagnosis must be a hitting set of all the conflicts (de Kleer and Williams 1987; Williams and Ragno 2007; Stern et al. 2012). Intuitively, since at least one component in every conflict is faulty, only a hitting set of all conflicts can explain the unexpected observation (failed test).

Barinel is a recently proposed software MBD algorithm (Abreu, Zoeteweij, and van Gemund 2011) based on exactly this concept: considering traces of tests with failed outcome as conflicts and returning their hitting sets as diagnoses. In addition, Barinel computes a score to every diagnosis it returns. This score estimates the likelihood that a given diagnosis is true. This score considers both failed and passed tests, assigning lower scores to diagnoses containing components that are in traces of passed tests. For the exact computation of this score see Abreu et. al. (2011).

The SFL-based approach to software diagnosis, and Barinel algorithm in particular, can scale well to large systems. However, it may output a large set of diagnoses, providing less guidance to developer tasked to fix the corresponding bug.

Test Planning in TDP

To prune the number of diagnoses, further tests should be performed. We propose next several algorithms for generating such additional tests. These algorithms are intended to be used in the TDP paradigm (Algorithm 1, line 3), where they are called iteratively until a single diagnosis is found. As such, we refer to these algorithms as *test planning* algorithms, emphasizing that the overall goal is to minimize the number of tests done until the correct diagnosis is found.

The input to all of the test planning algorithms below is $\langle OBS, \Omega, T \rangle$: OBS is the set of observed tests, Ω is the set of diagnoses produced by Barinel, and T is the set of tests that can be performed by the tester. For every observed test $t \in OBS$ we know $trace(t)$ and whether t passed or failed. For every diagnosis $\omega \in \Omega$, we know the score given to it by Barinel, and denote it by $p(\omega)$. These scores are normalized to one, i.e., $\sum_{\omega \in \Omega} p(\omega) = 1$, and we consider $p(\omega)$ to be an approximation of the probability that ω is true.

While we assume that the set of tests to perform is given, it is also possible to generate them using test generation algorithms (Fraser and Arcuri 2011; Campos et al. 2013). Our focus is on how to choose which tests the tester should perform and not on how to generate new possible tests. Recall, that for a human tester performing all the tests in T is costly, and we would like to find the correct diagnosis by performing minimal tests from T.

Before a test t is performed, its trace may not be known. Code analysis and past executions of t can estimate the possible traces t might have. We simplify this by considering $trace(t)$ to be known even before t is executed, and leave the research about the uncertainty of $trace(t)$ to future work. Next, we present several test planning algorithms to select the next test.

Best Diagnosis and Highest Probability

The *Best Diagnosis* (BD) test planning algorithm chooses randomly a test t from T whose trace includes a component that is part of the most likely diagnosis found so far. The intuition behind BD is to test the most likely diagnosis, potentially proving it to be correct and returning it. Formally, BD returns a random test from the set

$$\{t \in T | \exists \omega \in \Omega \;\; s.t. \;\; \omega \cap trace(t) \;\wedge\; p(\omega) = \max_{\omega' \in \Omega} p(\omega')\}$$

The *Highest Probability* (HP) test planning algorithm chooses tests that pass through the component that is most likely to be faulty. We denote by $p(C, \Omega)$ the likelihood that a component C is faulty based on Ω. $p(C, \Omega)$ is calculated as the sum over the scores of all the diagnoses that contain C, i.e., $p(C, \Omega) = \sum_{\omega \in \Omega, C \in \omega} p(\omega)$. HP returns a random test from the set

$$\{t \in T | \exists C \in COMPS \;\; s.t. \;\; C \in trace(t)$$
$$\wedge\, p(C, \Omega) = \max_{C' \in COMPS} \{p(C', \Omega) | p(C', \Omega) < 1\}\}$$

Components C with $p(C, \Omega) = 1$ are avoided as they are already known to be faulty. HP is motivated by the assumption that checking first components that are likely to be faulty will result in finding the correct diagnosis faster.

As an example of applying BD and HP, consider the observed tests given in Table 1. There are 3 components C_1, C_2 and C_3 in the system and 4 tests were performed $T_1, .., T_4$. Note that T_3 and T_4 produce different outcomes (T_3 fails while T_4 passes) while they have the same trace. This occurs in real software, where bugs can cause failure intermittently. Running Barinel on these observed tests results

Test	C_1	C_2	C_3	Failed?
T_1	1	1	0	Yes
T_2	0	1	1	Yes
T_3	1	0	1	Yes
T_4	1	0	1	No

Table 1: An example of Obs

in finding three diagnoses $\Delta_1 = \{C_1, C_2\}$, $\Delta_2 = \{C_2, C_3\}$, and $\Delta_3 = \{C_1, C_3\}$, with $p(\Delta_1) = p(\Delta_2) = 0.455$ and $p(\Delta_3) = 0.09$. Assume that there are 3 possible tests that can be performed, T_5, T_6, and T_7, each having a trace of a single component C_1, C_2, or C_3, respectively. The most likely diagnoses are Δ_1 and Δ_2 and thus BD would choose a test from $\{T_5, T_6, T_7\}$ randomly, as all tests pass through one of the components in Δ_1 or Δ_3. By contrast, HP would choose T_6, as $p(C_2, \Omega)$=0.91 is the most likely component to be faulty.

Entropy

A well-studied approach for test generation in the diagnosis literature and others is based on information metrics, attempting to identify the test that executing it will provide the most information about the tested system (Yang, Dang, and Fischer 2011). Common information metrics used in similar contexts are *information gain* and *entropy*, where the information gain of a test t is the difference between the entropy of the set of diagnoses before performing t and the expected entropy of the diagnoses after performing t. We suggest to compute the entropy of Ω with respect to the score of the diagnoses, i.e,. $Entropy(\Omega) = \sum_{\omega\in\Omega} -p(\omega)\cdot\log(p(\omega))$. See that if Ω has a diagnosis with score close to one, i.e., a diagnosis that is very likely to be correct, results in $Entropy(\Omega)$ that is close to zero, while having many diagnoses with a similar, low, score yields high entropy.

To calculate the information gain of performing a test t, we first estimate the probability of t passing w.r.t Ω, denoted as $p(t, \Omega)$, as follows:

$$p(t, \Omega) = \sum_{\omega\in\Omega} p(\omega) \cdot \prod_{C\in(\omega\cap trace(t))} goodness(C)$$

where $goodness(C)$ is a value computed by Barinel that estimates the likelihood of C causing a test passing through it to fail.

Depending on whether t will pass or fail, the set of diagnoses Ω will change. Let $\Omega_+(t)$ and $\Omega_-(t)$ be the resulting set of diagnoses assuming that t passes or fails, respectively. Using $p(t, \Omega)$, $\Omega_-(t)$, and $\Omega_+(t)$, we compute the information gain of performing a test t as follows.

$$\text{InfoGain}(t) = Entropy(\Omega) - (p(t, \Omega) \cdot Entropy(\Omega_+(t))$$
$$(1 - p(t, \Omega)) \cdot Entropy(\Omega_-)(t))$$

The corresponding test planning algorithm, which we simply call *Entropy*, chooses a random test from all tests with highest InfoGain($\cdot$).

Test Planning as an MDP

The test planning algorithms proposed above are myopic, in the sense that they do not perform any long-term planning of the testing process .[1] Next, we propose a non-myopic test planning algorithm that chooses tests by explicitly reasoning about their possible outcomes and subsequent testing. Specifically, we model test planning as a Markov Decision Process (MDP).

MDP Modeling An MDP is composed of states (including an initial state and possibly a set of terminal states), actions, a transition function, and a reward function. For test planning, a state is a set of observed tests (including traces and outcomes). The initial state is the set of initial tests already performed by tester (Obs). Before describing the rest of the MDP, observe that running Barinel on the observed tests of a state s returns the set of diagnoses that will be found if s is reached. We call this set of diagnoses the diagnoses of s, denoted by $\Omega(s)$.

The actions in the MDP correspond to the set of tests T. The outcome of an action is either $pass(t)$ or $fail(t)$. Thus, the state transition function corresponds to the probability that a planned test t will pass given the observed tests in state s, estimated by $p(t, \Omega(s))$ (as described for the Entropy test planning algorithm). A state with $|\Omega(s)| = 1$ is a state where Barinel produces a single diagnosis. In such cases TDP halts and that diagnosis is passed to the developer. Thus, such states are terminal states. Our MDP is a shortest path MDP, where we seek the shortest sequence of tests to reach a terminal state. Hence, we set the reward function $R(s, a)$ to be minus one for every non-terminal state s.

Reaching a state where Ω contains a single diagnosis can be impossible or very time consuming. We consider a weaker requirement – performing tests until there is a diagnosis that is very likely to be true according to its score. Formally, we define $p_{max}(s) = \max_{\omega\in\Omega} p(\omega)$ and every state s with $p_{max}(s)$ higher than a threshold B is regarded as a terminal states. In our experiments we set B to be 0.9.

Solving the MDP A solution to an MDP is a policy mapping states to actions. An MDP solver seeks the policy that maximizes the expected reward that will be gained when executing that policy. In our case, an MDP solver seeks the policy that minimizes the expected number of tests until the correct diagnosis is found. There are many algorithms for solving MDPs such as Value Iteration, Policy Iteration (Russell and Norvig 2010), and Real-Time Dynamic Programming (Barto, Bradtke, and Singh 1995). An optimal solution to our MDP would result in an optimal test plan.

The number of states in this MDP is doubly exponential in the number of available tests (a state for every possible set of tests and their outcome). As a result, the corresponding MDP is too large to be solved optimally by current optimal MDP solvers, as they usually require time at least linear in the number of states. Thus, we used a suboptimal MDP solver based on Sparse Sampling (SpS) (Kearns, Mansour, and Ng 2002), a well known MDP solver specifically designed for large state space. In SpS, the expected reward of performing an action a at a state s is evaluated by sampling C possible outcomes of performing a and recursively esti-

[1]One might argue that Entropy is semi-myopic, as it considers the outcome of a test to compute the entropy after it is executed.

mating the expected reward of these outcomes until reaching a predefined depth H. C and H are parameters of the algorithm called *width* and *depth*, respectively. SpS requires a *generative model* to define how to sample the outcomes of performing an action, i.e., the states reached by performing a at state s. In our case, the generative model simply samples the outcome of the considered test.

To improve efficiency, we modified SpS in two ways:

Using a default policy. In SpS, in every state s every possible action is sampled C times. Since the number of tests, and correspondingly actions, may be large, we only sampled all the tests in the initial state, and sampled a single test in subsequent states. This can be viewed as a *default policy*, a concept successfully used in UCT (Kocsis and Szepesvári 2006) and other sampling-based algorithms (Nguyen, Lee, and Leong 2012). In our experiments we used all the proposed heuristics (HP, BD, and Entropy) as well as simple random test selection as default heuristics.

Values of leaf states. In SpS, every sample halts when it either reaches a terminal state or reaches depth H. We experimented with two methods for setting the value of leaf states so as to estimate the amount of further testing required to reach a terminal state. The first method we experimented with estimated the value of a leaf state by performing a *rollouts*: choosing further tests to perform according to the default policy and selecting test outcomes randomly until reaching a terminal state, and propagating backwards the cost of reaching that terminal state. That cost is the value of the corresponding leaf state. While effective, the rollout method is computationally intensive, and we were only able to apply it to the smallest benchmark we used (the vending machine program described later). For larger benchmarks, we used instead a linear interpolation prediction of the expected cost of finding a state with $p_{max} \geq B$, starting from a leaf state s. For an initial state s_{init} this is computed by $-\frac{H\cdot(B-p_{max}(s))}{p_{max}(s)-p_{max}(s_{init})}$.

There are many variants of SpS and other MDP solvers that can be used, as well as other ways to estimate the reward of leaf states. The focus of this work is not to propose a general purpose MDP solver, but to present an effective MDP solver that, as we demonstrate in the experimental results section, results in an effective test planning algorithm.

Experimental Results

To evaluate TDP and the proposed heuristics we performed experiments on three benchmarks. In every experiment we run all the proposed test planning algorithms as part of TDP. An experiment was halted when one of the following conditions was met: 1) a diagnosis with probability higher than 0.9 was found, and 2) if all tests have been executed. The second condition exists because there are cases where it is not possible to identify a single diagnosis with probability higher than 0.9. In such a case, no further testing will help, and the set of diagnoses will be passed to the developer.

Vending Machine Experiments

The first benchmark we used, inspired by the "vending machine" benchmark (Campos et al. 2013; Burger and Zeller 2011; Orso et al. 2006), is a simple implementation of a

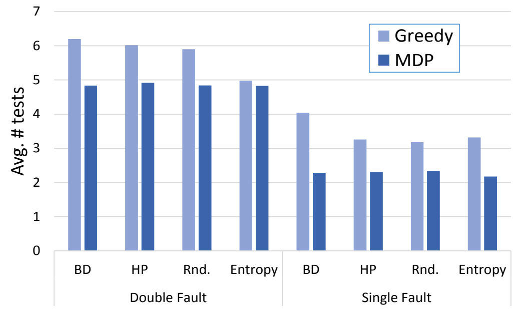

Figure 2: Vending machine results.

vending machine logic having 19 components and 240 lines of code. We injected potential bugs in four components and hand crafted 15 possible tests (T). We partition the results to single and double fault instances, where one or two potential bugs were activated, respectively. For every choice of activated bugs we run 100 instances, where each instance chose randomly 7 out of the 15 possible test to serve as the observed tests that were initially performed by the tester before invoking TDP.

Figure 2 shows the average number of tests (y-axis) for every test planning algorithm. The darker blue lines correspond to an MDP test planning algorithm, using HP, BD, Entropy or random (denoted "Rnd.") as a default policy. As can be seen, for both single and double fault instances, the number of tests required until TDP halted was substantially smaller when using the MDP tests planning algorithm. In addition, the MDP results were robust over the evaluated default policies, showing similar performance for all the different default policies.

In the single fault cases, all non-MDP test planning algorithm (the light blue bars) show very similar results, except for BD performing substantially worse. The bad performance of BD is understandable, as it does not distinguish between the components in the best diagnosis. The benefit of Entropy over HP, BD, and random is only observable in the double fault instances, where Entropy required less tests on average (5 vs. ≈6). Note that the average number of tests required for the double faults instance was, in general, much larger for all test planning algorithm, demonstrating that the double fault instances are harder to solve. These results suggest that Entropy's benefit over even simple random testing is needed for the harder instances. Also note that in these cases, the benefit of MDP over Entropy is not large.

Experiments on Randomly Generated Programs

In the benchmark above the possible tests and bugs were hand crafted to simulate reasonable bugs and tests. As a result, the size of the diagnosed program was relatively small. To perform experiments on larger programs, we generated synthetic programs, bugs, and tests, as follows.

First, we generated random graphs with 300 nodes, where every two nodes are connected by an edge with probability of 1.3%. From every graph $G = (V, E)$ we generated a program such that every node $v \in V$ corresponded to a function, and an edge $(v_1, v_2) \in E$ correspond to a function call, i.e.,. function v_1 calls v_2. The neighbors of every node v are ordered and grouped into one to four groups (number

	MDP		Entropy		HP	
Prob.	10	20	10	20	10	20
0.5	20.1	24.9	56.2	107.0	71.3	130.8
0.6	23.8	25.6	57.8	111.0	71.8	132.7
0.7	25.8	26.8	65.1	111.9	74.3	133.9
0.8	26.0	32.2	69.9	115.3	77.6	135.3
0.9	26.9	33.0	73.0	143.9	79.0	136.4

Table 2: TDP costs for the randomly generated graphs.

Prob.	MDP	Entropy	HP
0.5	5.08	50.75	66.75
0.6	6.45	53.85	70.14
0.7	6.45	53.85	70.14
0.8	6.45	53.85	70.14
0.9	6.45	55.69	70.41

Table 3: TDP costs for the NUnit call graph.

of groups is random between 1 and 4). The ordering correspond to the order of execution, and the grouping correspond to conditional choices of execution.

We generated 100 such synthetic programs, each generated from a different graph that was randomly generated as described above. For each of these synthetic programs we chose randomly 10 functions and injected a bug in them. One of the nodes in the graph was selected to be the entry point of the program, and paths in the graph starting from the entry point corresponded to tests. The set of possible tests T was chosen to be the shortest paths from the entry point to each of nodes in the graph. For each of the 100 problem instances, we chose 15 tests randomly to serve as the initially observed tests. We also generated similar instances with 20 injected bugs. To manage the experiment runtimes, we halted TDP for cases where more than 160 TDP iterations were needed.

For each of the problem instances, we run TDP with each of the proposed test planning methods. We omit the results of random test selection and BD as both performed worse than HP, Entropy, and MDP. Following MDP's observed robustness over the different default policies, we show only the results for MDP with the random default policy. In contrast to the small vending machine benchmark above, here tests vary significantly in the size of their trace. As our purpose is to save tester effort in finding the correct diagnosis, we assumed here that the cost of performing a test is proportional to the number of components in its trace.

Table 2 shows the average cost required to find a diagnosis having a score higher than X, for $X = 0.5, ..., 0.9$. The columns headed by 10 and 20 represent results for instances with 10 and 20 bugs, respectively. Similar to the vending machine benchmark, here too the results show that diagnosing instances with more bugs is more costly for all algorithms, and that the MDP-based algorithm performed significantly better than all other algorithms.

Experiments on Open Source Software

In the next set of experiments we extracted the call graph from an open source project called NUnit (www.nunit.org). NUnit is a well-known testing framework for the .NET programming languages (e.g, C#), which is an adaptation of JUnit, a well-known testing framework for Java. Using built-in tools in MS Visual Studio, we extracted a call graph from the source code of the NUnit "core" project, version 2.4. The resulting graph had 302 nodes. Cycles were removed from the resulting graph and a random number of nodes were set to be faulty. This random number was chosen from the values $\{10, 15, 20, 25, 30\}$. As in the previous benchmark, the available tests (T) were the shortest paths in the graph to every component, and the initial observed tests were 15 randomly selected tests.

Table 3 shows the average cost of finding a diagnosis having score higher than X for $X = 0.5, .., 0.9$ using the proposed test planning algorithms. Every number in the table is an average over 110 tests. The same trends discussed in the random graph experiment were also observed here. Entropy and HP show substantially worse results than MDP. Note that in general the cost of finding a diagnosis of a given score was smaller in the NUnit graph than in the randomly generated graphs, even though they were roughly the same size ($\approx$300 nodes). We conjuncture that this is attributed to the topology of the NUnit call graph, which is more tree-like than the generated random graphs.

In general, in all three benchmarks MDP outperformed all other methods significantly, emphasizing the benefit of non-myopic planning in TDP. Naturally, the MDP method is the most computationally intensive method. The runtime of the other test planning algorithms were negligible in all our experiments, while the runtime of the MDP method was kept manageable via the use of SpS. Importantly, the purpose of a test planning algorithm is not to reduce computational complexity but to minimize the number of tests performed by a human tester. Thus, experimenting with more efficient state-of-the-art MDP solvers is beyond the scope of this paper.

Conclusion and Future Work

In this paper we proposed the TDP testing paradigm, where the tester, enhanced with AI techniques, identifies for the developer the faulty software component that caused the bug. TDP is built from two components: an MBD algorithm and a test planning algorithm. The MBD algorithm suggests a set of diagnoses and the test planning algorithm plans further tests to provide information for identifying the correct diagnosis. For the MBD part of TDP, we propose to use Barinel (Abreu, Zoeteweij, and van Gemund 2011), which is an SFL-based MBD algorithm that can scale to large systems without using any modeling of the diagnosed software. For the test planning part of TDP, we proposed the BD, HP, Entropy and MDP-based test planning algorithms. Evaluation on three domains suggest that the MDP-based test planning algorithm performs substantially better than all other test planning algorithms.

This paper presents only the first building block of this vision: automated diagnosis and automated test planning. In future work we plan to perform an empirical evaluation 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.

Acknowledgments

We thank the Kamin program and the Israeli Science Foundation grant number 182/13 for partially funding this research.

References

Abreu, R.; Zoeteweij, P.; and van Gemund, A. J. C. 2009. Spectrum-based multiple fault localization. In *Automated Software Engineering (ASE)*, 88–99. IEEE.

Abreu, R.; Zoeteweij, P.; and van Gemund, A. J. C. 2011. Simultaneous debugging of software faults. *Journal of Systems and Software* 84(4):573–586.

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

Burger, M., and Zeller, A. 2011. Minimizing reproduction of software failures. In *International Symposium on Software Testing and Analysis*, 221–231. ACM.

Campos, J.; Abreu, R.; Fraser, G.; and d'Amorim, M. 2013. Entropy-based test generation for improved fault localization. In *Automated Software Engineering (ASE), IEEE/ACM*, 257–267.

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

Esser, M., and Struss, P. 2007. Fault-model-based test generation for embedded software. In *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI'07, 342–347.

Fraser, G., and Arcuri, A. 2011. Evosuite: automatic test suite generation for object-oriented software. In *SIGSOFT FSE*, 416–419.

González-Sanchez, A.; Abreu, R.; Groß, H.-G.; and van Gemund, A. J. C. 2011. An empirical study on the usage of testability information to fault localization in software. In *SAC*, 1398–1403.

Kearns, M.; Mansour, Y.; and Ng, A. Y. 2002. A sparse sampling algorithm for near-optimal planning in large markov decision processes. *Machine Learning* 49(2-3):193–208.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*. Springer. 282–293.

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

Nguyen, T.-H. D.; Lee, W.-S.; and Leong, T.-Y. 2012. Bootstrapping monte carlo tree search with an imperfect heuristic. In *Machine Learning and Knowledge Discovery in Databases*. Springer. 164–179.

Orso, A.; Joshi, S.; Burger, M.; and Zeller, A. 2006. Isolating relevant component interactions with jinsi. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, 3–10. ACM.

Perez, A.; Abreu, R.; and Riboira, A. 2014. A dynamic code coverage approach to maximize fault localization efficiency. *Journal of Systems and Software*.

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

Silva, J. 2011. A survey on algorithmic debugging strategies. *Adv. Eng. Softw.* 42(11):976–991.

Stern, R.; Kalech, M.; Feldman, A.; and Provan, G. M. 2012. Exploring the duality in conflict-directed model-based diagnosis. In *AAAI*.

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

Williams, B. C., and Ragno, R. J. 2007. Conflict-directed A* and its role in model-based embedded systems. *Discrete Appl. Math.* 155(12):1562–1595.

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

Yang, L.; Dang, Z.; and Fischer, T. R. 2011. Information gain of black-box testing. *Formal aspects of computing* 23(4):513–539.

Zeller, A. 2002. Isolating cause-effect chains from computer programs. In *SIGSOFT FSE*, 1–10.