Type of Submission

choose one of:	
original research paper: this is original unpublished work and we confirm that this submission is different enough from our previously published work	✓
PhD panel submission: this manuscript describes the work I have been conducting, or am planning to conduct in the course of my PhD thesis	
resubmission: this is a resubmission of a previously !published! paper, but we would like to present it at *DX'17*	
for original research papers only: choose one of	
yes, we want our paper to be published in the official *DX'17* proceedings	
no, we do not want our paper to be published in the official *DX'17* proceedings, only a 300 word abstract	✓
for resubmissions only:	
(a) specify some information (see notes below!)	
the original version of this paper was already published at:	
—-please enter the full conference/workshop name + abbreviation here	
link to the electronic version of the original paper (if available):	
`e.g., http://ijcai.org/Proceedings/13/Papers/160.pdf`	
link to an electronic author version of the original paper (if available):	
`e.g., http://www.ist.tugraz.at/pill/downloads/PillQuaritsch.pdf`	
(b) choose one of	
yes, we would like a 300 word abstract of our presentation to appear in the front matter of the proceedings	
no, there shall be no abstract in the front matter of the proceedings, but only a notice of our presentation	

Learning Software Behavior for Automated Diagnosis

Ori Bar-Ilan and **Roni Stern** and **Meir Kalech**
Software and Information Systems Engineering
The Cyber Security Research Center at Ben Gurion University of the Negev
Be'er Sheva, Israel

Abstract

Software diagnosis algorithms aim to identify the faulty software components that caused a failure. A key challenges of existing software diagnosis algorithms is how to prioritize the outputted diagnoses. To do so, previous work proposed a method for estimating the likelihood that each diagnosis is correct. Computing these diagnosis likelihoods is non-trivial. We propose to do this by learning a behavior model of the software components and use it to identify abnormally behaving components. In this work we show the potential of such an approach by performing an empirical evaluation on a synthetic behavior model of the components. The results show that even an imperfect behavior model is useful in improving diagnosis accuracy and minimizing wasted troubleshooting efforts.

Introduction

Software diagnosis is the task of identifying the root cause of an observed software bug. That is, the task in software diagnosis is to identify the software components – classes, functions, or lines of code – that needs to be fixed in order to prevent the bug from occurring in the future. The need for an automated tool able to perform software diagnosis grows as modern software systems are highly complex. Indeed, software bugs are prevalent in virtually all software products and their impact can be catastrophical.

Model-based diagnosis (MBD) is a principled approach for automated diagnosis that has also been proposed for software diagnosis [16]. In MBD, a model of the system is needed along with observations of the system's behavior. These observations are checked against the given model, and inference algorithms are used to produce *diagnoses*, which are possible assumptions about which components are faulty that are consistent with the given model and the observations.

Software systems can rarely be modeled accurately and thus directly applying MBD to software diagnosis is difficult. To this end, Abreu et al. [3] proposed Barinel, a software diagnosis algorithm that finds diagnoses by analyzing previously executed tests, their traces, and their outcomes (pass or fail). Notably, Barinel does not require any modeling of the software components' behaviors.

Barinel may output a large set of candidate diagnosis, but only one diagnosis is actually correct. Conveniently, Barinel also assigns a value to each candidate diagnosis that is roughly associated with the likelihood that it is correct. The importance of this likelihood estimation has led several prior works to focus exactly on improving this likelihood function [3; 2; 4].

In this preliminary work, we set out to investigate the potential of a novel approach for improving the accuracy of the previously proposed methods to compute the likelihood function. The approach we propose is based on learning an approximate behavior model of the software components. This learning is done over observations of the inputs and outputs of the program's components, collected while the system is being tested.

To evaluate the potential of this approach, we conducted an empirical evaluation of over several open source projects. In this evaluation we used a synthetic behavior model, whose accuracy is controlled by a parameter. Then, compared the diagnostic accuracy – which is directly affected by the accuracy of the diagnosis likelihood function – of the vanilla Barinel and the Barinel that uses our modified diagnosis likelihood function. The results show that our approach has huge potential to increase Barinel's diagnostic accuracy, even with a reasonable amount of error in the components' behavior model.

Preliminaries

A program is composed of a set of components $COMPS = \{C_1, \ldots, C_M\}$. For the ongoing discussion, assume that every component is a method in the program, but our exposition also holds for different component granularity, e.g., where the component is a class or program statement.

Every software component C has a set of input and output variables, denoted $in(C)$ and $out(C)$.[1] The *behavior* of component C, denoted $\Phi(C)$, describes the relation between $in(C)$ and $out(C)$. That is, we expect that if all components follow their healthy behavior then all tests will pass. If a component does not follow its healthy behavior, we refer to it as *faulty*. The *health* predicate $h(C)$ denotes that component C has followed its healthy behavior.

Classical MBD is designed for cases where observations of the system are inconsistent with the assumption that all components follow their healthy behavior. More formally, classical MBD is defined by $\langle COMPS, SD, OBS \rangle$ where OBS is the set of observations and SD is a formal description of the system's behavior, usually in the form of a set

[1] If the program has a state that affects the behavior of C then it is also part of $in(C)$, and similarly, if C affects this state then it will be in $out(C)$.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1M} \\ a_{21} & a_{22} & \dots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \dots & a_{NM} \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix}$$

Figure 1: Input to SFL.

of behaviors for the components in COMPS. Under this formalism, an MBD problem arises when $SD \wedge OBS$ is inconsistent with the assumption that all components are healthy. Diagnosis algorithms aim to find a set of components ω such that $SD \wedge OBS$ is consistent with the assumption that all components are healthy except those in ω. ω is referred to as a *diagnosis* and is defined formally as any set of components that satisfy the following.

$$SD \wedge OBS \wedge \bigwedge_{C \in COMPS \setminus \omega} h(C) \wedge \bigwedge_{C \in \omega} \neg h(C)$$

Many MBD algorithms use *conflicts* to direct the search towards diagnoses.

Definition 1 (Conflict)**.** *A set of components $\gamma \subseteq$ COMPS is a conflict if $\bigwedge_{C \in \gamma} h(C) \wedge$ SD $\wedge$ OBS is inconsistent.*

Conflicts are useful because every diagnosis must be a hitting set of all the conflicts since at least one component in every conflict is faulty [8; 18; 13]. Indeed, MBD algorithms such as GDE and CDA* find diagnoses by finding conflicts and considering only their hitting sets.

Spectrum-based Software Diagnosis

Software systems are usually too large and dynamic to allow a formal specification of the software components' behavior, and thus assuming that Φ is given, is not practical. Instead, we assume that a set of tests has been executed and we are given information about these tests. In particular, for each test t we are given its *outcome* (pass or fail), and its *trace*, i.e., the set of components involved in that test. We denote the outcome and trace of t by $outcome(t)$ and $trace(t)$, respectively. Note that from a practical perspective, it is easy to modify a set of tests so that they output all the above information, e.g. by instrumenting the source code using Java's JVMTI.

Abreu et al. [3] proposed Barinel, a software diagnosis algorithm that uses this information to find candidate diagnoses. The key concept in Barinel is that if a test failed then at least one of the components in its trace is faulty. Thus, the trace of a failed test is a conflict (Definition 1), and Barinel considers it as such when computing diagnoses. Then, it uses a fast hitting set algorithm called STACATTO [1] to find hitting sets of these conflicts, which are then outputted as diagnoses.

Barinel provides an elegant bridge between MBD and Spectrum-based Fault Localization (SFL). The input to most SFL algorithms is in the form of two boolean matrices. The first matrix, A, is of size $N \times M$ and it describes the presence or absence of each of the M components in the traces of each of the N tests, having $a_{ij} = 1$ *iff* component c_j is in the trace of test t_i. The second matrix, E, which is of size $N \times 1$, contains the outcome (pass/fail) for each of the N tests ($e_i = 1$ *iff* t_i failed). The matrix E is used as OBS, and SD is represented by the connection between the traces and the observation:

$$\forall i \in [0, N] : \left(\bigwedge_{j \in [0,M], a_{ij}=1} h(C_j) \right) \rightarrow (e_i = 0)$$

The main drawback of using Barinel is that it often outputs a large set of diagnoses, thus providing weaker guidance to the programmer that is assigned to solve the observed bug. To address this problem, Barinel computes a *score* for every diagnosis it returns, estimating the likelihood that it is true. This serves as a way to prioritize the large set of diagnoses returned by Barinel.

The score of diagnosis ω reflects its probability to be correct $Pr(\omega)$. Since only the observations are known, the probability of a diagnosis to be correct (i.e. to describe the actual system faults) depends solely on the degree of which ω explains the observations. Thus, it is required to compute $Pr(\omega|OBS)$. Applying Bayes' rule achieves the following:

$$Pr(\omega|OBS) = \frac{Pr(OBS|\omega)}{Pr(OBS)} \cdot Pr(\omega) \qquad (1)$$

$Pr(OBS)$ does not depend on ω thus it can be considered as a normalizing factor. $Pr(\omega)$ is the prior probability of ω . If this probability is unknown, it can be computed using this formula: $Pr(\omega) = p^{|\omega|} \cdot (1-p)^{M-|\omega|}$, where p denotes the *a priori* probability that a component is faulty. In case that no such information is available, a uniform distribution can be applied. $Pr(OBS|\omega)$ is defined as follows:

$$Pr(OBS|\omega) = \begin{cases} 0, & \text{if } OBS \wedge \omega \text{ are inconsistent} \\ 1, & \text{if } OBS \text{ is unique to } \omega \\ \epsilon, & \text{otherwise} \end{cases} \qquad (2)$$

$Pr(OBS|\omega)$ gets the value of 0 or 1 only in rare cases. ϵ is often referred to as *epsilon policy* and it is where the actual reasoning of the diagnoses ranking relies. There are many epsilon policies [7; 2]. Barinel offers one of those. Barinel makes use of the probability of a faulty component C_j to produce a correct output. This probability is denoted by η_j. Formally,

$$\eta_j = Pr(e(t) = 0 | C_j \in trace(t) \wedge \neg h(C_j)) \qquad (3)$$

Assuming that η_j is known for each of the components, ϵ can now be computed as follows:

$$\epsilon = \begin{cases} \prod_{C_j \in \omega_i \wedge a_{ij}=1} \eta_j, & \text{if } e_i = 0 \\ 1 - \prod_{C_j \in \omega_i \wedge a_{ij}=1} \eta_j, & \text{if } e_i = 1 \end{cases} \qquad (4)$$

Intuitively, if a test t_i passed, then all of the faulty components in $trace(t_i)$ (i.e. faulty components that participated in t_i) would have produced a valid output. Therefore, the result is the product of their probability to produce a valid output, even though they are faulty. Since η_j is unknown (for all components), it is estimated by using the maximum likelihood technique which maximizes the diagnosis' probability of being accurate.

Barinel's score function relies on these goodness functions to estimate the probability that a component produced an incorrect output thus caused a test to fail. In this work we propose a novel approach for computing these goodness

function. Our new approach yields more accurate diagnoses as it leverages the information available by the tests' probes (those are the input and output values of the components in the trace), and it is much faster to compute than the maximum likelihood estimates originally used in Barinel.

Learning Components' Behavior

In this work we assume the existence of *probes* that monitor the input and output values of components involved in traces of tests. Implementing such probes can be done in a similar way to the way used to collect the test traces, e.g., via Java's JVMTI instrumentation capabilities. The input and output values of component C observed by the probes of test t are denoted by $in(t, C)$ and $out(t, C)$, respectively.

With these probes, we aim to replace the goodness function η_j with a per-test goodness function, denoted b_{ij}.

$$b_{ij} = Pr(outcome(t) = 1 | C_j \in trace(t) \wedge \neg h(C_j) \wedge (in(t, C_j) = in(t_i, C_j)) \wedge (out(t, C_j) = out(t_i, C_j)) \quad (5)$$

In words, b_{ij} is the probability of a test t to fail given that component C_j is in its trace and it is faulty and its input and output values are the input and output values observed for C_j in test t_i. We call b_{ij} the *"probe probability"* of component C_j in test T_i, to highlight that, unlike Barinel, it specifically uses the information in the probes. Next, we show how to estimate b_{ij} and how it can be used to create a new epsilon policy.

Estimating the Probe Probabilities

In order to learn the value of b_{ij}, we propose to use Machine Learning techniques. In particular, we propose to learn a binary classifier for predicting the outcome of a test given the probes of a single component in that test. I.e., for a given test t and a component C, we aim to learn a classifier that accepts $in(t, C)$ and $out(t, C)$ and predicts $outcome(t)$.

Machine learning proceeds by building a training set to train a classification model (a classifier) and then evaluate it with a test set. The training set consists a set of instances, each with its classification label (the correct classification). Each instance consists of a set of features.

In our settings, the features are the input and output values observed for a component in a test, and the classification label is whether that test has passed or failed. So, to build the training set we obtain a sample of $\langle in(t, C), out(t, C), outcome(t) \rangle$ tuples, e.g., by running all of the program's test suites. This training set is then given as an input to a Machine Learning binary classification algorithm that trains a classifier that determines if a sample of $in(C)$ and $out(C)$ correlates strongly with a test failure or a test success.

Importantly, binary classifiers often output a *confidence* level, stating the classifier's confidence about the label it has outputted. For example, when using tree ensembles, confidence can be measured by the number of individual trees that voted for the predicted class, out of all trees. We use these confidence values as our probe probabilities.

Using the Probe Probabilities

The collection of all predictive models is combined into the input matrix for the Spectrum-based Fault Localization algorithm, and used to compute the goodness function. This is done by adding the probability b_{ij} ($\forall i, j \in \mathbb{N}$ such that $1 \leq i \leq N \wedge 1 \leq j \leq M$). This results in a new transformed input, as seen in Figure 2.

$$\begin{bmatrix} a_{11}, b_{11} & a_{12}, b_{12} & \dots & a_{1M}, b_{1M} \\ a_{21}, b_{21} & a_{22}, b_{22} & \dots & a_{2M}, b_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1}, b_{N1} & a_{N2}, b_{N2} & \dots & a_{NM}, b_{NM} \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix}$$

Figure 2: Transformed input to SFL.

To compute ϵ, we assume, as Barinel, that software components fail independently of each other, and thus the probability of b_{ij} and b_{kl} are independent. Using this independence property, if a test t_i fails, we can compute ϵ by multiplying the probe probabilities of all components that are diagnosed as faulty and participated in t_i. In case a test passes, we use the complementary probability to the probe probability of each component that is diagnosed as faulty.

Our novel goodness function, notated with ϵ', is computed as such:

$$\epsilon' = \begin{cases} \prod_{C_j \in \omega \wedge a_{ij}=1} 1 - b_{ij}, & \text{if } e_i = 0 \\ \prod_{C_j \in \omega \wedge a_{ij}=1} b_{ij}, & \text{if } e_i = 1 \end{cases} \quad (6)$$

Experimental Results

In this section we present semi-synthetic experiments which show the potential benefit of our approach. We start by describing how we setup the experiment and then we present the results.

Experiment Setup

Project	Versions	Tests	Methods	Bugs Reported	Bugs Fixed
Orient	44	790	19,207	4,625	2,459
CDT	231	3,990	66,982	17,713	9,091
ANT	72	5,190	10,830	5,890	1,176
POI	72	2,346	21,475	3,361	1,408

Table 1: Details on the projects used for benchmarking.

As a benchmark, we used the four real-world open-source projects used by Elmishali et al. (9). Table 1 lists fundamental statistics about these projects. Each project has its own source code files, tests and bug reports. The tests of each project were run and used to produce the input required by our algorithm, i.e., the tests' traces, probes observations, and outcomes. The bug reports were used to create the ground truth diagnosis, i.e., the actual components that are faulty. Using the benchmark above does not only help us measuring our success in a real-world scenario, but also to compare it to recent state-of-the-art diagnosers.

Synthesizing Probe Probabilities

To be able to measure the potential of the use of probe probabilities in SFL, we used *synthetic* probe probabilities in our experiments instead of real probe probabilities that were learned from observed tests. These synthetic probe probabilities were generated with respect to the ground truth diagnosis (i.e. the set of faulty components), denoted ω_{gt}, and an

error parameter, denoted err. ω_{gt} and err were used to synthesize probe probabilities b'_{ij} using the following formula:

$$b'_{ij} = \begin{cases} 0, & \text{if } e_i = 0 \\ 0, & \text{if } e_i = 1 \wedge a_{ij} = 0 \\ err, & \text{if } e_i = 1 \wedge a_{ij} = 1 \wedge C_j \notin \omega_{gt} \\ 1 - err, & \text{if } e_i = 1 \wedge a_{ij} = 1 \wedge C_j \in \omega_{gt} \end{cases} \quad (7)$$

The first case describes a scenario where a test did not fail. The second case describes a scenario where a test did fail but C_j did not participate in that test. In both cases, $b_{ij} = 0$. The third case describes a scenario where a test failed, C_j participated in that test, and C_j is known to be healthy. In the most ideal scenario, our probe probability would be zero, indicating that while test i failed, it was not caused by component C_j and so C_j must not appear in any diagnosis. Introducing *err* instead of 0, simulates the possible error in the our learned probe probability. The fourth case describes a similar scenario to the third case, only that C_j is now known to be faulty. In the most ideal scenario, our probe probability would be 1, indicating that C_j is faulty and thus promoting diagnoses that contain it.

Evaluated Diagnosers

Throughout our evaluation process, we assessed the diagnoser we present in this paper against two other state-of-the-art diagnosers. The first diagnoser is Barinel, which was previously introduced. The second diagnoser, devised by Elmishali et al. [9], presents a data-augmented improvement to Barinel [9]. This improvement is done by learning various meta-data features of open sourced projects (via version control, issue tracker etc) and produce more accurate a-priori probabilities for components to be faulty. This algorithm will be referred to as *DA* during our evaluation. Our diagnoser is referred to as *Probe*.

Evaluation Metrics

Each of the evaluated diagnosers outputs one or more diagnoses, each associated with a likelihood score. To compare the outputs of the different algorithms we used the following metrics.

Weighted mean precision and recall

The first two metrics we used are the *weighted mean precision* and *weighted mean recall*, which were previously introduced by Elmishali et al. (9). To compute these metrics, we first computed the precision and recall for every diagnosis outputted by the evaluated diagnoser. The weighted mean precision is then computed as the weighted average over the precision of all diagnoses, using the diagnoses' score as its weight. The weighted mean recall is computed in a similar manner, averaging the diagnoses' recall scores. The benefit of these metrics is that they aggregate precision and recall over the result set of diagnosis, with consideration to the score assigned by the diagnoser to each diagnosis. So, we expect higher weighted mean precision and weighted mean recall for diagnosers that output more accurate likelihood scores. For brevity, we will refer to both weighted mean precision and weighted mean recall as simply precision and recall, respectively.

Health state wasted effort

The health state has recently presented by Stern et al. (14; 15). The health state indicates the probability of each component to be faulty, given a set of diagnoses Ω and a probability function over them p:

$$H(C) = \sum_{\omega \in \Omega} p(\omega) \cdot \mathbb{1}_{C \in \omega} \quad (8)$$

where $\mathbb{1}_{C \in \omega}$ is the indicator function defined as:

$$\mathbb{1}_{C \in \omega} = \begin{cases} 1 & C \in \omega \\ 0 & \text{otherwise} \end{cases}$$

The wasted effort metric estimates the effort wasted in repairing healthy components assuming that components are repaired in a decreasing order of their health state probability ($H(C)$) until all faulty components are repaired. Ties were broken randomly. The wasted effort is the number of healthy components that are repaired, normalized by the number of healthy components.

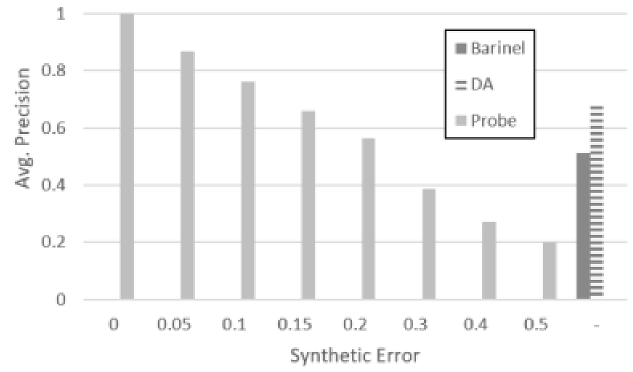

Figure 3: Weighted mean precision.

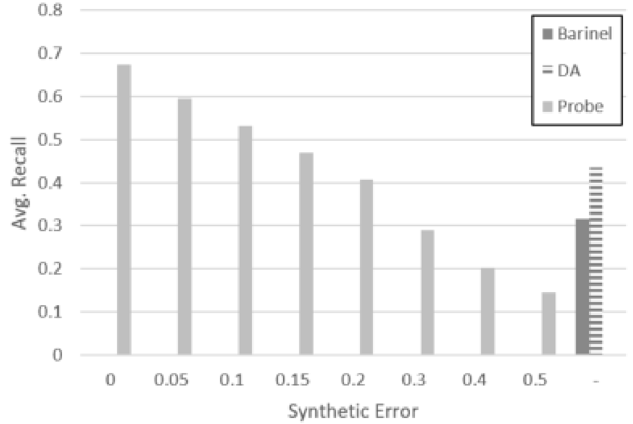

Figure 4: Weighted mean recall.

Results

In Figure 3 and Figure 4 we present the precision and recall metrics, respectively. The x-axis represents the different synthetic error rates. It is clear that the Probe diagnoser, with 0.15 synthetic error, achieves similar results as the DA diagnoser. Also, the Probe diagnoser, with 0.2 synthetic error, achieves significantly better results than Barinel.

In Figure 5 we present a comparison of the Health state wasted effort between the same diagnosers. It is visible that even with 0.3 synthetic error rate, the Probe diagnoser, achieves superior results over both the DA and Barinel diagnosers.

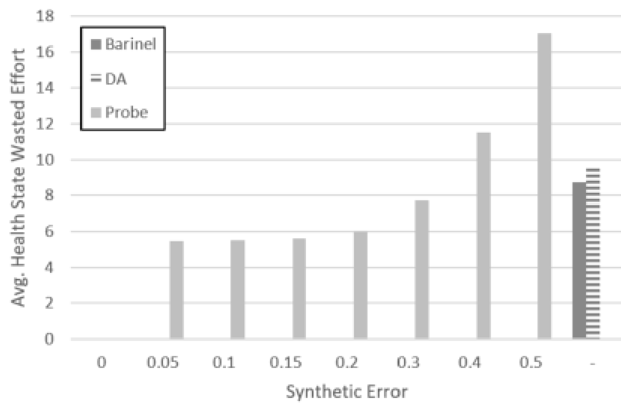

Figure 5: Health state wasted effort.

The key challenge, left for future work, is to learn probe probabilities that has sufficiently low prediction errors. However, our results are very promising, demonstrating that the error required for our approach to be useful is very reasonable – ranging from 0.3 to 0.15 (depending on the evaluated metric).

Related Work

Hofer and Wotawa introduced a new approach, Spectrum ENhanced DYnamic Slicing [11; 10], which combines Spectrum Fault Localization (SFL) with slicing hitting set computation [19]. The method they propose computes slices for all faulty variables in all failing test cases. A slice is a subset of a program which behaves like the original program for a given set of variables [17]. Then the diagnoses are computed by a hitting set algorithm based on Reiter's HS-tree [12]. The SFL approach assists to compute the fault probabilities based on both the failed tests as well as the passed tests. The advantage of combining SFL and SHSC lies in the use of slicing approach to distinguish statements occurring in the same basic building block, as well as analyzing the execution information from both passing and failing test cases as done in SFL.

In a recent work Cardoso et al. (5) extend Barinel to deal with degradation failures in software. In many cases the existence of a fault is linked to degradation of quality of service, rather than a clear case of a failed test. Cardoso et al. propose a fuzzy logic framework to model the observation and extend the computation of the likelyhood function Barinel applies to deal with fuzzy observations.

Elmishali et al. (9) propose a data-driven approach to better prioritize the set of diagnoses returned by Barinel. In particular, they use methods from the software engineering literature to learn from collected data how to predict which software components are expected to be faulty. Then, they integrate these predictions into Barinel to better prioritize the diagnoses it outputs and provide more accurate diagnosis likelihood estimates.

Improving the likelihood function by learning an approximate behavior model, as proposed in this paper, is orthogonal to all the above works. We demonstrate its benefits in relation to SFL, but in the same manner it might assist to Spectrum ENhanced DYnamic Slicing, degradation failures and the fault prediction.

Conclusion and Future Work

In this paper we proposed a novel way to compute the probability of a test to fail given information on the inputs and outputs of the components involved in the test. We claimed that this probabilities can be learned using a machine learning technique. Then, we showed how these probabilities can be used to improve the way Barinel, a state-of-the-art software diagnosis algorithm, ranks diagnoses. Our preliminary results, demonstrate the potential of our approach for increasing Barinel's diagnostic accuracy.

In future work we will demonstrate the applicability of our approach by replacing the synthetic probabilities we used in this work with probe probabilities learned from the software's demonstrated behavior while executing tests. Furthermore, since this technique is orthogonal to other improvements to Barinel discussed in the related work section, we plan to develop a hybrid software diagnosis algorithm that combines these techniques, harnessing the complementary strengths of each technique. Lastly, it is important to note an inherent difficulty that lies within our datasets: Data Imbalance. Imbalance datasets are frequent in many applications [6]. This is also true for our application. Naturally, most of the tests in a test suite pass, while few fail. This creates an imbalance within our datasets since most of the records will contain one out of the two classes (the one that correlates with passed tests).

References

[1] R. Abreu and A. J. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *SARA*, volume 9, pages 2–9, 2009.

[2] R. Abreu and A. J. C. van Gemund. Diagnosing multiple intermittent failures using maximum likelihood estimation. *Artif. Intell.*, 174(18):1481–1497, 2010.

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

[4] N. Cardoso and R. Abreu. A kernel density estimate-based approach to component goodness modeling. In M. desJardins and M. L. Littman, editors, *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*. AAAI Press, 2013.

[5] N. Cardoso, R. Abreu, A. Feldman, and J. de Kleer. A framework for automatic debugging of functional and degradation failures. In G. A. Kaminka, M. Fox, P. Bouquet, E. Hüllermeier, V. Dignum, F. Dignum, and F. van Harmelen, editors, *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, pages 569–576. IOS Press, 2016.

[6] S. Cateni, V. Colla, and M. Vannucci. A method for resampling imbalanced datasets in binary classification tasks for real-world problems. *Neurocomputing*, 135:32–41, 2014.

[7] J. de Kleer. Diagnosing intermittent faults. In *Proceedings of International Workshop on Principles of Diagnosis (DX'07)*, 2007.

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

[9] A. Elmishali, R. Stern, and M. Kalech. Data-augmented software diagnosis. In D. Schuurmans and M. P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 4003–4009. AAAI Press, 2016.

[10] B. Hofer and F. Wotawa. Spectrum enhanced dynamic slicing for better fault localization. In *ECAI*, pages 420–425, 2012.

[11] B. Hofer, F. Wotawa, and R. Abreu. AI for the win: improving spectrum-based fault localization. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–8, 2012.

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

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

[14] R. Stern, M. Kalech, S. Rogov, and A. Feldman. How many diagnoses do we need? In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 1618–1624. AAAI Press, 2015.

[15] R. Stern, M. Kalech, S. Rogov, and A. Feldman. How many diagnoses do we need? *Artificial Intelligence*, 2017.

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

[17] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, July 1982.

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

[19] F. Wotawa. Fault localization based on dynamic slicing and hitting-set computation. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 161–170. IEEE, 2010.