Data-Augmented Software Diagnosis

Amir Elmishali, Roni Stern and **Meir Kalech**
Ben Gurion University of the Negev
Be'er Sheva, Israel

Abstract

Software fault prediction algorithms predict which software components is likely to contain faults using machine learning techniques. Software diagnosis algorithm identify the faulty software components that caused a failure using model-based or spectrum based approaches. We show how software fault prediction algorithms can be used to improve software diagnosis. The resulting data-augmented diagnosis algorithm overcomes key problems in software diagnosis algorithms: ranking diagnoses and distinguishing between diagnoses with high probability and low probability. We demonstrate the efficiency of the proposed approach empirically on three open sources domains, showing significant increase in accuracy of diagnosis and efficiency of troubleshooting. These encouraging results suggests broader use of data-driven methods to complement and improve existing model-based methods.

Introduction

Software is prevalent in practically all fields of life, and its complexity is growing. Unfortunately, software failures are common and their impact can be very costly. As a result, there is a growing need for automated tools to identify software failures and isolate the faulty software components, such as classes and functions, that have caused the failure. We focus on the latter task, of *isolating faults in software components*, and refer to this task as software diagnosis.

Model-based diagnosis (MBD) is an approach to automated diagnosis that uses a model of the diagnosed system to infer possible *diagnoses*, i.e., possible explanations of the observed system failure. While MBD was successfully applied to a range of domains (Williams and Nayak 1996; Feldman et al. 2013; Struss and Price 2003; Jannach and Schmitz 2014), it has not been applied successfully yet to software. The reason for this is that in software development, there is usually no formal model of the developed software. To this end, a scalable software diagnosis algorithm called Barinel has been proposed (Abreu, Zoeteweij, and van Gemund 2011). Barinel is a combination of MBD and Spectrum Fault Localization (SFL). SFL considers traces of executions, and finds diagnoses by considering the correlation between execution traces and which executions have failed. While very scalable, Barinel suffers from one key disadvantage: it can return a very large set of possible diagnoses for the software developer to choose from. To handle this disadvantage, Abreu et al. (2011) proposed a Bayesian approach to compute a likelihood score for each diagosis. Then, diagnoses are prioritized according to their likelihood scores.

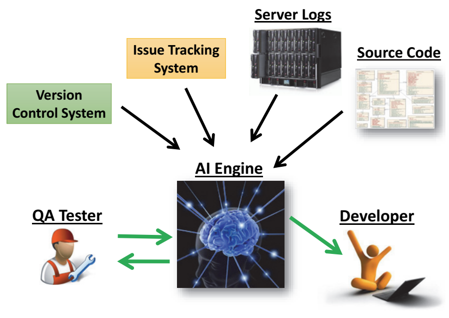
Thanks to the open source movement and current software engineering tools such as version control and issue tracking systems, there is much more information about a diagnosed system than revealed by the traces of performed tests. For example, version control systems store all revisions of every source files, and it is quite common that a bug occurs in a source file that was recently revised. Barinel is agnostic to this data. We propose a data-driven approach to better prioritize the set of diagnoses returned by Barinel.

In particular, we use methods from the software engineering literature to learn from collected data how to predict which software components are expected to be faulty. Then, we integrate these predictions into Barinel to better prioritize the diagnoses it outputs and provide more accurate diagnosis likelihood estimates.

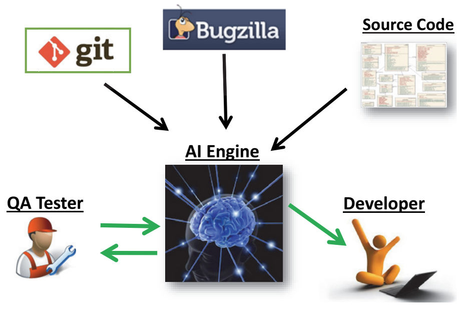
The resulting data-augmented diagnosis algorithm is part of a broader software troubleshooting paradigm that we call *Learn, Diagnose, and Plan (LDP)*. In this paradigm, illustrated in Figure 1(a), the troubleshooting algorithm learns which source files are likely to fail from past faults, previous source code revisions, and other sources. When a test fails, a data-augmented diagnosis algorithm considers the observed failed and passed tests to suggest likely diagnoses leveraging the knowledge learned from past data. If further tests are necessary to determine which software component caused the failure, such test are planned automatically, taking into consideration the diagnoses found. This process continues until a sufficiently accurate diagnoses is found.

We implemented this paradigm and evaluated its execution on a popular open source software projects. In particular, we demonstrated how the required data can be extracted from common software engineering tools like the Git version control and the Bugzilla issue tracking systems was used, as illustrated in Figure 1(b) and explained in the experimental results.

Results show a huge advantage of using our data-augmented diagnoser over Barinel with uniform priors for both finding more accurate diagnoses and for better select-



(a) Learn, Diagnose, and Plan Paradigm



(b) Our current implementation

Figure 1: The learn, diagnose, and plan paradigm and our implementation.

ing tests for troubleshooting. Moreover, to demonstrate the potential benefit of our data-augmented approach we also experimented with a synthetic fault prediction model that correctly identifies the faulty component. As expected, using the synthetic fault prediction model is better than using the learned fault prediction model, thus suggesting room for further improvements in future work. To our knowledge, this is the first work to integrate successfully a data-driven approach to a software diagnosis algorithm.

Related Work

Hofer and Wotawa introduced a new approach, Spectrum ENhanced DYnamic Slicing (Hofer, Wotawa, and Abreu 2012; Hofer and Wotawa 2012), which combines SFL with slicing hitting set computation (Wotawa 2010). 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 (Weiser 1982). Then the diagnoses are computed by a hitting set algorithm based on Reiter's HS-tree (Reiter 1987). 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. Learning the prior probabilities, as proposed in this paper, is orthogonal to the fault localization approach. We demonstrate its benefits in relation to SFL, but in the same manner it might assist to Spectrum ENhanced DYnamic Slicing.

Faults in spreadsheets is a good example for the using of software fault localization methods (Jannach et al. 2014). Hofer et al. (2013) explicitly proposed to adapt spectrum-based fault-localization from the traditional programming domain to spreadsheets. In particular they use SFL to compute the fault probabilities of the spreadsheet cells. Abreu et. al. (2015) propose a constraint-based approach for debugging spreadsheets based on the user expectations. Diagnosis candidates are explanations for the misbehavior in user expectations. Hofer and Wotawa (2014) present a comparison between a value-based approach and dependency-based approach and show that the later is much faster than the value-based approach. We believe that, as we show in software diagnosis, learning the prior fault probabilities of the cells in spreadsheets may improve the diagnosis accuracy.

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*

$$\bigwedge_{C\in\Delta}(\neg h(C)) \wedge \bigwedge_{C'\notin\Delta}(h(C')) \wedge SD \wedge OBS$$

is consistent, i.e., if assuming that the components in Δ *are faulty, then* SD *is consistent with* OBS*.*

The set of components ($COMPS$) in software diagnoses can be, for example, the set of classes, or all functions, or even a component per line of code. Low level granularity of components, e.g., setting each line of code as a component, will result in very focused diagnoses (e.g., pointing on the exact line of code that was faulty). Focusing the diagnoses in such way comes at a price of an increase in the computational effort. Automatically choosing the most suitable level of granularity is a topic for future work.

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 are that it does not scale well and modeling the behavior of software component is often infeasible.

SFL for Software Diagnosis

An alternative approach to software diagnosis has been proposed by Abreu et al. (2009; 2011; 2012; 2013), 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 running 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 applications maintain a log with some form of this information.

In the SFL-based approach, SD is implicitly defined 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 then we can infer that 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 $\bigwedge_{C \in \Gamma} h(C) \wedge SD \wedge OBS \models \bot$*

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. With a fast hitting set algorithm, such as the STACATTO hitting set algorithm proposed by Abreu et al. (2009), Barinel can scale well to large systems. 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.

Prioritizing Diagnoses

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 exact details of how this score is computed is given by Abreu et al. (2009; 2011). For the purpose of this paper, it is important to note that the score computation used by Barinel is Bayesian: it computes for a given diagnosis the posterior probability that it is correct given the observed passes and failed tests. As a Bayesian approach, Barinel also requires some assumption about the *prior probability* of each component to be faulty. Prior works using Barinel has set these priors uniformly to all components. In this work, we propose a data-driven way to set these priors more intelligently and demonstrate experimentally that this has a huge impact of the overall performance of the resulting diagnoser.

Data-Augmented Software Diagnosis

The prior probabilities used by Barinel represent the a-priori probability of a component to be faulty, without considering any observed system behavior. Fortunately, there is a line of work on *software fault prediction* in the software engineering literature that deals exactly with this question: which software components are more likely to have a bug. We propose to use these software fault predictions as priors to be used by Barinel. First, we provide some background on software fault prediction.

Software Fault Prediction

Fault prediction in software is a classification problem. Given a software component, the goal is to determine its class – healthy or faulty. Supervised machine learning algorithms are commonly used these days to solve classification problems. They work as follows. As input, they are given a set of *instances*, in our case these are software components, and their correct labeling, i.e., the correct class for each instance. In our case, the class is whether the software component is healthy or faulty. They output a *classification model*, which maps an instance to a class.

Learning algorithm extract *features* from a given instance, and try to learn from the given labeled instances the relation between the features of an instance and its class. A key to the success of machine learning algorithms is the choice of *features* used. Many possible features were proposed in the literature for software fault prediction.

Radjenovic et al. (2013) surveyed the features used by existing software prediction algorithms and categorizes them into three families. **Traditional.** These features are traditional software complexity metrics, such as number of lines of code, McCabe (1976) and Halstead (1977) complexity measures.

Object Oriented. These features are software complexity metrics that are specifically designed for object oriented programs. This includes metrics like cohesion and coupling levels and depth of inheritance.

Process. These features are computed from the software change history. They try to capture the dynamics of the software development process, considering metrics such as lines added and deleted in the previous version and the age of the software component.

In a preliminary set of experiments we found that the combination of features that performed best is a combination of 115 features from the features listed by Radjenovic et al. (2013) worked best. This list of features included the Mc-

Cabe (1976) and Halstead (1977) complexity measures, several object oriented measures such as the number of methods overriding a superclass, number of public methods, number of other classes referenced, and is the class abstract, and several process features such as the age of the source file, the number of revisions made to it in the last release, the number of developers contributed to its development, and the number of lines changed since the latest version.

As shown in the experimental results section, the resulting fault prediction model was accurate enough so that the overall data-augmented software diagnoser be more effective than Barinel with uniform priors. However, we are sure that a better combination of features can be found, and this can be a topic for future work. The main novelty of our work is in integrating a software fault prediction model into the software fault localization process.

Integrating the Fault Prediction Model

The software fault prediction model generated as described above is a classifier, accepting as input a software component and outputting a binary prediction: is the component predicted to be faulty or not. Barinel, however, requires a real number that estimates the prior probability of each component to be faulty. To estimated priors from the fault prediction model, we rely on the fact that most prediction models also output a confidence score, indicating the model's confidence about the classified class. Let $\mathit{conf}(C)$ denote this confidence for component C. We use $\mathit{conf}(C)$ for Barinel's prior if C is classified as faulty, and $1 - \mathit{conf}(C)$ otherwise.

Obtaining a Training Set

For both learning and testing a fault prediction model, we require a mapping between reported bug and the source files that were faulty and caused it. Manually tracking past bugs and tracing back their root cause is clearly not a scalable solution. Fortunately, most projects these days use a version control system and an issue tracking system. Version control systems, like Git and Mercurial, track modifications done to the source files. Issue tracking systems, like Bugzilla and Trac, record all reported bugs and track changes in their status, including when a bug gets fixed. A key feature in modern issue tracking and version control systems is that they enable tracking which modification to source files are done in order to fix a specific bug.

Of course, not all files modified when a bug is fixed actually caused the bug. For example, some bug fixes include adding a parameter to a function, consequently modifying all places where that function is called although they could not be regarded as "faulty" source files. As a crude heuristic to overcome this, we considered for a given bug X only the source file whose revision were most extensive among all files associated with fixing bug X. Śliwerski et al. (2005) proposed a more elaborate method to heuristically identify the source files that are caused the bug, when analyzing a similar data set. Being able to automatically generate a training set highlights one of the main advantages of our work: it can be applied to any software project that uses a version control and issue tracking system. Thus, our approach can be readily deployed in such cases.

Project	Start	Ver.	Files	Bugs
CDT (*eclipse.org/cdt*)	2002	231	8,750	9,091
POI (*poi.apache.org*)	2002	72	2,810	1,408
Ant (*ant.apache.org*)	2000	72	1,195	1,176

Table 1: Details on the domains we experimented on.

Experimental Results

We implemented and evaluated the proposed data-augmented approach as follows. As a benchmark, we used the source files and bugs reported for three popular open-source projects: (1) Eclipse CDT, which is an IDE for C/C++ that is part of the Eclipse platform, (2) Apache Ant, which is a build tool, and (3) Apache POI, which provides a Java API for Microsoft documents. All projects are written in Java. Table 1 lists in the columns "Start", "Ver.", "Files", and "Bugs" when the project started, number of version so far (including minor version), number of Java source files, and number of bugs reported and fixed, respectively.

Project	Precision	Recall	F-Measure	AUC
ANT	0.206	0.175	0.189	0.775
CDT	0.550	0.086	0.149	0.846
POI	0.280	0.252	0.265	0.845

Table 2: Evaluating the fault prediction models.

All three projects use the Git version control system and the Bugzilla issue tracking system. For each project, we used the last version as a test set and the 4 versions before it as a training set. The first set of results we report is the quality of our fault prediction model on these three benchmarks. The Weka software package (*www.cs.waikato.ac.nz/ml/weka*) was used to experiment with several learning algorithms. In a preliminary comparison we found the Random forest learning algorithm to perform best in our domains. Table 2 shows the precision, recall, F-measure, and AUC of the fault prediction models generated by Random forest (with 1,000 trees) for each of the benchmark projects. Precision, recall, F-measure, and AUC are standard metrics for evaluating classifiers. In brief, *precision* is the ratio of faulty files among all files identified by the evaluated model as faulty. *Recall* is the number of faulty files identified as such by the evaluated model divided by the total number of faulty files. *F-measure* is a known combination of precision and recall. The AUC metric addresses the known tradeoff between recall and precision, where high recall often comes at the price of low precision. This tradeoff can be controlled by setting different sensitivity thresholds to the evaluated model. AUC is the area under the curve plotting the accuracy as a function of the recall (every point is a different threshold value). All metrics range between zero and one (where one is optimal) and are standard metrics in machine learning and information retrieval. The unfamiliar reader can find more details in Machine Learning books, e.g. Mitchell's classical book (Mitchell 1997).

As the results show, while AUC scores are reasonable, the precision and especially recall results are fairly low. This is understandable, as most files are not faulty, and thus the training set is very imbalanced. An imbalanced data set is a known inhibitor of the performance of standard learning algorithms. We have experimented with several known methods to handle imbalanced datasets, such as SMOTE and random under sampling, but these did not produce substantially better results. However, as we show below, even this imperfect prediction model is able to improve existing data-agnostic software diagnosis algorithm.

Diagnosis Performance

Next, we evaluated our data-augmented diagnoser when using the fault prediction model evaluated above. The input is a set of tests, with their traces and outcomes and the output is a set of diagnoses, each diagnosis having a score that estimates its correctness. This score was computed by Barinel as described earlier in the paper, where the data-agnostic diagnoser uses uniform priors and the proposed data-augmented diagnoser uses the predicted fault probabilities from the learned model.

Each of the software projects we used includes a test package that contains automated tests that can be run automatically. For every experiment, we chose a package and a known bug that occurred in this package. Then, we choose a subset of the automated tests that test this package and ran them to record their trace. Test outcome was simulated by assuming that a bug that is in a trace has a probability of 0.2 to fail. This is added because a nature behavior of a software faulty component is that it does not always cause tests passing through it to fail.

To compare the set of diagnoses returned by the different diagnosers, we computed the *weighted average* of their precision and recall as follows. First, the precision and recall of every returned diagnosis was computed. Then, we averaged the precision and recall of all the returned diagnoses, weighted by the score given to the diagnoses by Barinel, normalized to one. This enables aggregating the precision and recall of a set of diagnoses while incorporating which diagnoses are regarded as more likely according to Barinel. For brevity, we will refer to this weighted average precision and weighted average recall as simply precision and recall.

The rows "Agn." and "Aug." in Table 3 show the average precision and recall for the data-agnostic and data-augmented diagnosers, respectively. The rows "Syn(0.1)" and "Syn(0.0)" will explained be later. A problem instance consists of (1) a bug, and (2) a set of observed tests, chosen randomly, while ensuring that at least one test would pass through the faulty files. We experimented with 10, 20, 30, and 40 observed tests (along with their traces and outcomes), corresponding to the columns of Table 3. Each result in the table is an average over the precision and recall obtained for 50 problem instances.

Both precision and recall of the data-augmented and data-agnostic diagnosers support the main hypothesis of this work: a data-augmented diagnoser can yield substantially better diagnoses that a data-agnostic diagnoser. For example, the precision of the data-augmented diagnoser for the POI project with 20 tests is 0.76 while it is only 0.60 for the data-augnostic diagnoser. No clear trend is observed from adding additional tests.

POI	Precision				Recall			
Tests	10	20	30	40	10	20	30	40
Agn.	0.56	0.60	0.61	0.61	0.51	0.54	0.54	0.55
Aug.	0.66	0.76	0.64	0.64	0.61	0.68	0.56	0.56
Syn.(0.1)	0.70	0.81	0.77	0.77	0.62	0.70	0.65	0.65
Syn.(0.0)	0.79	0.98	0.88	0.88	0.66	0.83	0.73	0.73
CDT	Precision				Recall			
Tests	10	20	30	40	10	20	30	40
Agn.	0.60	0.49	0.46	0.45	0.52	0.41	0.38	0.38
Aug	0.64	0.59	0.55	0.56	0.54	0.46	0.41	0.42
Syn.(0.1)	0.89	0.81	0.69	0.69	0.89	0.85	0.68	0.68
Syn.(0.0)	0.99	0.98	0.83	0.83	0.80	0.70	0.59	0.58
ANT	Precision				Recall			
Tests	10	20	30	40	10	20	30	40
Agn.	0.58	0.68	0.68	0.68	0.48	0.49	0.49	0.49
Aug.	0.66	0.73	0.73	0.73	0.56	0.53	0.53	0.53
Syn.(0.1)	0.79	0.86	0.86	0.86	0.69	0.66	0.66	0.66
Syn.(0.0)	0.97	0.99	0.99	0.99	0.87	0.79	0.79	0.79

Table 3: Avg. recall and precision for diagnostic task.

Synthetic Priors Building better fault prediction models for software is an active field of study (Radjenovic et al. 2013) and thus future fault prediction models may be more accurate than the ones used by our data-augmented diagnoser. To evaluate the potential benefit of a more accurate fault prediction model on our data-augmented diagnoser, we created a *synthetic fault prediction model*, in which faulty source files get P_f probability and healthy source files get P_h, where P_f and P_h are parameters. Setting $P_h = P_f$ would cause the data-augmented diagnoser to behave in a uniform distribution exactly like the data-agnostic diagnoser, setting the same prior probability for all source files to be faulty. By contrast, setting $P_h = 0$ and $P_f = 1$ represent an optimal fault prediction model, that exactly predicts which files are faulty and which are healthy.

The lines marked "Syn. (X)" in Table 3 mark the performance of the data-augmented diagnoser when using this synthetic fault prediction model, where $X = P_h$ and $P_f = 0.6$. Note that we experimented with many values of P_f and P_h, and presented above a representative subset of these results.

As expected, setting lowering the value of P_h results in more better diagnoses being found. Setting a very low P_h value improves the precision significantly up to almost perfect precision in some cases (0.99 for ANT when given 20 observed tests). Thus clearly demonstrating the potential benefit of the proposed data-augmented approach.

Troubleshooting Task

Efficient diagnosers are key components of *troubleshooting algorithms*. Troubleshooting algorithms choose which tests to perform to find the most accurate diagnosis. Zamir et al. (2014) proposed several troubleshootings algorithms specifically designed to work with Barinel for troubleshooting software bugs. In the below preliminary study, we evaluated the impact of our data-augmented diagnoser on the overall performance of troubleshooting algorithms. Specifically, we implemented the so-called *highest probability* (HP) troubleshooting algorithm, in which tests are chosen in the following manner. HP chooses a test that is expected to pass through the source file having the highest probability of being faulty, given the diagnoses probabilities.

We run the HP troubleshooting algorithm with each of the diagnosers mentioned above (all rows in Table 3). We compared the HP troubleshooting algorithm using different diagnosers by counting the number of tests required to reach a diagnosis of score higher than 0.7. Note that the aim of a troubleshooting algorithm is to minimize the number of required tests. Our hypothesis is that a data-augmented diagnoser will reduce the number of tests since the learned prior probabilities of the software components might direct the HP troubleshooting algorithm to select tests which pass through components with higher probability.

POI	Steps				Precision			
Tests	10	20	30	40	10	20	30	40
Agn.	5.0	9.6	17.4	20.4	0.52	0.53	0.55	0.52
Aug.	4.4	7.4	9.8	17.4	0.63	0.69	0.68	0.55
Syn.(0.1)	2.8	4.0	10.0	8.6	0.62	0.71	0.69	0.72
Syn.(0.0)	0.2	0.0	0.0	1.0	0.74	0.91	0.88	0.85
CDT	Steps				Precision			
Tests	10	20	30	40	10	20	30	40
Agn.	7.0	16.0	27.0	31.0	0.57	0.44	0.42	0.42
Aug.	5.2	8.4	15.8	13.0	0.66	0.58	0.52	0.53
Syn.(0.1)	2.0	7.4	14.4	18.6	0.74	0.73	0.72	0.76
Syn.(0.0)	0.0	0.0	0.4	0.0	0.92	0.88	0.84	0.87
ANT	Steps				Precision			
Tests	10	20	30	40	10	20	30	40
Agn.	5.4	7.8	8.6	8.0	0.53	0.58	0.57	0.60
Aug.	4.0	6.6	6.0	6.8	0.57	0.67	0.64	0.67
Syn.(0.1)	3.6	5.0	5.6	4.6	0.70	0.71	0.71	0.75
Syn.(0.0)	0.0	0.0	0.0	0.0	0.89	0.92	0.92	0.92

Table 4: Avg. steps and precision for troubleshooting.

Table 4 is structured in a similar way to Table 3 except that it shows the average number of tests performed by the HP troubleshooting algorithm until it halts (in the "Steps" columns) and the precision of the found diagnosis (in the "Precision" columns). The results show the same over-arching theme: the data-augmented diagnoser is much better than the data-agnostic diagnoser for this troubleshooting task, being able to halt earlier with a higher quality diagnosis. Also, using the synthetic fault prediction model can result in even further improvement, thus suggesting future work for improving the learned fault prediction model.

Conclusion, and Future Work

We incorporated a software fault prediction model into the software diagnosis algorithm Barinel (Abreu, Zoeteweij, and van Gemund 2009). The resulting data-augmented diagnoser is shown to outperform Barinel without such a fault prediction model. This was verified experimentally using three open source projects. Results also suggests that future work on improving the learned fault prediction model will result in an improved diagnosis accuracy. In addition, it is worthwhile to incorporate the proposed data-augmented diagnosis methods with other proposed improvements of the based SFL-based software diagnosis, as those proposed by Hofer et al. (2012; 2012).

References

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

Abreu, R.; Hofer, B.; Perez, A.; and Wotawa, F. 2015. Using constraints to diagnose faulty spreadsheets. *Software Quality Journal* 23(2):297–322.

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.

Campos, J.; Abreu, R.; Fraser, G.; and d'Amorim, M. 2013. Entropy-based test generation for improved fault localization. In Denney, E.; Bultan, T.; and Zeller, A., eds., *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, 257–267. IEEE.

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

Feldman, A.; de Castro, H. V.; van Gemund, A.; and Provan, G. 2013. Model-based diagnostic decision-support system for satellites. In *IEEE Aerospace Conference*, 1–14. IEEE.

Halstead, M. H. 1977. *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc.

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

Hofer, B., and Wotawa, F. 2014. On the usage of dependency-based models for spreadsheet debugging. *Software Engineering Methods in Spreadsheets*.

Hofer, B.; Riboira, A.; Wotawa, F.; Abreu, R.; and Getzner, E. 2013. On the empirical evaluation of fault localization techniques for spreadsheets. In Cortellessa, V., and VarrÃş, D., eds., *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 68–82.

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

Jannach, D., and Schmitz, T. 2014. Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. *Automated Software Engineering* 1:1–40.

Jannach, D.; Schmitz, T.; Hofer, B.; and Wotawa, F. 2014. Avoiding, finding and fixing spreadsheet errors - A survey of automated approaches for spreadsheet QA. *Journal of Systems and Software* 94:129–150.

McCabe, T. J. 1976. A complexity measure. *IEEE Trans. Software Eng.* 2(4):308–320.

Mitchell, T. 1997. *Machine learning*. McGraw Hill.

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

Radjenovic, D.; Hericko, M.; Torkar, R.; and Zivkovic, A. 2013. Software fault prediction metrics: A systematic literature review. *Information & Software Technology* 55(8):1397–1418.

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

Śliwerski, J.; Zimmermann, T.; and Zeller, A. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30(4):1–5.

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

Struss, P., and Price, C. 2003. Model-based systems in the automotive industry. *AI magazine* 24(4):17–34.

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

Williams, B. C., and Nayak, P. P. 1996. A model-based approach to reactive self-configuring systems. In *Conference on Artificial Intelligence (AAAI)*, 971–978.

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.

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

Zamir, T.; Stern, R.; and Kalech, M. 2014. Using model-based diagnosis to improve software testing. In *AAAI Conference on Artificial Intelligence*.