DeBGUer: A Tool for Bug Prediction and Diagnosis

Amir Elmishali, Roni Stern, Meir Kalech
The Cyber Security Research Center at Ben Gurion University of the Negev
amir9979@gmail.com, {sternron,kalech}@bgu.ac.il

Abstract

In this paper, we present the DeBGUer tool, a web-based tool for prediction and isolation of software bugs. DeBGUer is a partial implementation of the Learn, Diagnose, and Plan (LDP) paradigm, which is a recently introduced paradigm for integrating Artificial Intelligence (AI) in the software bug detection and correction process. In LDP, a diagnosis (DX) algorithm is used to suggest possible explanations – diagnoses – for an observed bug. If needed, a test planning algorithm is subsequently used to suggest further testing. Both diagnosis and test planning algorithms consider a fault prediction model, which associates each software component (e.g., class or method) with the likelihood that it contains a bug. DeBGUer implements the first two components of LDP, bug prediction (Learn) and bug diagnosis (Diagnose). It provides an easy-to-use web interface, and has been successfully tested on 12 projects.

Introduction

Software is ubiquitous and its complexity is growing. Consequently, software bugs are common and their impact can be very costly. Therefore, much effort is diverted by software engineers to **detect, isolate, and fix bugs** as early as possible in the software development life cycle.

Learn, Diagnose, and Plan (LDP) is a recently proposed paradigm that aims to support bug detection and isolation using technologies from the Artificial Intelligence (AI) literature (Elmishali, Stern, and Kalech 2018). To support bug detection, LDP uses Machine Learning to train a fault prediction model that predicts which software components are likely to contain bugs, providing guidance to testing efforts. To support bug isolation, LDP uses a software diagnosis algorithm to detect the root cause of failed tests, and techniques from automated planning to plan additional tests.

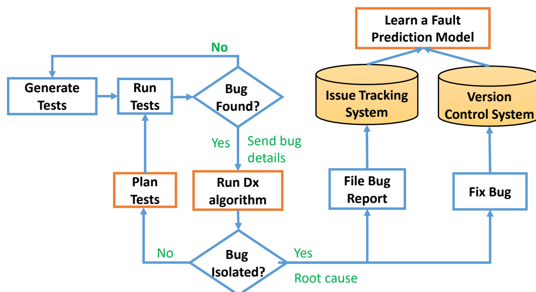

Figure 1: The workflow of the LDP paradigm.

In this paper, we describe DeBGUer, a novel open source tool that is a partial implementation of LDP. DeBGUer provides a web interface to the LDP bug prediction and diagnosis capabilities. To use DeBGUer, a developer simply needs to provide a link to its issue tracker (e.g., Jira) and source control management system (e.g., Git). Then, DeBGUer visualizes the fault prediction results, showing a fault (bug) likelihood estimate for every software component in a given version of the code. Through its web interface, DeBGUer also exposes the LDP diagnosis functionality: the developer points to a set of tests, DeBGUer executes them, and if some of the tests fail it outputs a set of software components that may have caused the tests to fail.

Our tool can be seen as a counterpart to the GZoltar tool (Campos et al. 2012). GZoltar provides two key functionalities: test suite minimization and software diagnosis. DeBGUer does not deal with test suite minimization. The DeBGUer software diagnosis algorithm is an improved version of diagnosis algorithm used by GZoltar.[1] DeBGUer also provides a software fault prediction functionality, which is not supported by GZoltar.

Having such an implementation of LDP – DeBGUer – serves several purposes. First, developers can use it in practice to follow LDP. Second, researchers working on software fault prediction or software diagnosis can use DeBGUer to evaluate their algorithms against the corresponding algorithms implemented in DeBGUer.

In this paper, we provide a brief description of the LDP paradigm and describe our implementation of it in DeBGUer. Then, we report on an empirical evaluation of DeBGUer on 12 open-source projects so far. Results show accurate bug prediction and isolation capabilities, demonstrating the applicability of LDP in general and DeBGUer specifically.

[1]GZoltar uses the Barinel algorithm (Abreu, Zoeteweij, and van Gemund 2009), while DeBGUer uses a data-augmented version of Barinel that were already shown to outperform the vanilla

Copyright © 2019, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[TIKA-1222] Tika does not extract attachments from RFC822 files Created: 16/Jan/14 Updated: 16/Mar/15 Resolved: 16/Mar/15

Status:	Resolved
Project:	Tika
Component/s:	parser
Affects Version/s:	1.4, 1.5, 1.6
Fix Version/s:	None

Figure 2: Export from the Jira issue tracker, showing data on bug TIKA-1222.

The LDP Paradigm

As a preliminary, we briefly describe the LDP paradigm. For a complete description please see Elmishali et al. (2018). LDP uses three AI components:

- **A fault predictor.** An algorithm that estimates the probability of each software component to have a bug.
- **A diagnoser.** A diagnosis (DX) algorithm that accepts as input a set of test executions including at least one failed test. It outputs one or more possible explanations for the failed tests. Each of these explanations, referred to as *diagnoses*, is an assumption about which software components are faulty. The diagnoser also associates every diagnosis with a score that indicates how likely the diagnosis is the true explanation for the failed tests.
- **A test planner.** An algorithm that accepts the set of diagnoses outputted by the diagnoser and suggests additional tests to perform in order to gain additional diagnostic knowledge that will help identifying the correct diagnosis.

Figure 1 illustrates the LDP workflow. The process starts by extracting information from issue tracking system (ITS) and version control system (VCS) that is used. ITS records all reported bugs and their status (e.g., fixed or open). VCS records every modification done to the source code. Prominent examples of ITS and VCS are Jira and Git, respectively. The information extracted from these systems, such as the set of past failures and code changes, is used by LDP to train a bug prediction model using standard Machine Learning.

When one or more tests fail, indicating that a bug exists, the set of executed tests is inputted to the LDP diagnoser. The diagnoser outputs a set of possible diagnoses and their score. If a single diagnosis is found whose score is high enough (how much is "high enough" is a parameter), then this diagnosis is passed to the developer to be fixed. If not, then the test planner proposes an additional test to be performed in order to narrow down the set of possible diagnoses. This initiates an iterative process in which the test planner plans an additional test, the tester performs it, and the diagnoser uses the new data obtained to compute the set of possible diagnoses and their likelihoods. This process stops when a diagnosis is found whose probability of being correct is high enough. At this stage a bug report is added to the issue tracking system and the diagnosed bug is given to a developer to fix the (now isolated) bug.

Barinel (Elmishali, Stern, and Kalech 2016).

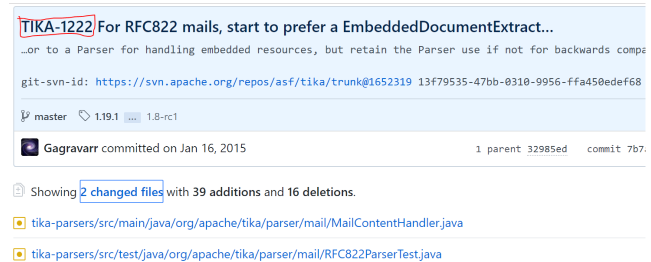

Figure 3: Screenshot from Github, showing the commit that fixed Bug TIKA-1222 from Figure 2.

Implementation Details

In this section, we describe the architecture of DeBGUer. As introduced, DeBGUer is partial implementation of the LDP paradigm. It includes AI components that perform two tasks: fault prediction and diagnosis. Next we describe the implementation details of these components as well as the web interface.

Fault Predictor

Fault prediction in software is a classification problem. Given a software component, such as method or class etc., the goal is to determine its class – healthy or faulty. Supervised machine learning algorithms are commonly used to solve classification problems. They work as follows. As input, they are given a set of *labeled instances*, which are pairs of instances and their correct labeling, i.e., the correct class for each instance. In our case, instances are software components such as classes or methods, and the labels are which software component is healthy and which is not. They output a *classification model*, which maps an (unlabeled) instance to a class. This set of labeled instances is called the *training set* and the act of generating a classification model from the training set if referred to as *learning*.

Learning algorithms extract *features* of the instances in the training set, and learn the relation between the features of the instances and their 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 features used by existing software prediction algorithms. In a preliminary set of experiments we found that the combination of features that performed best is a combination of 405 features from the features listed by Radjenovic et al. (2013) and bug history features that were created by us. This list of features includes the McCabe 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.[2]

[2]The exact list of the 405 features that we used can be

Description of the project

example of diagnosing apache reef with DeBGUer

Project name

reef

Project url

https://github.com/apache/reef

Issue tracker product name (jira)

REEF

Issue tracker url

https://issues.apache.org/jira

Versions to learn from.

reef-project-0.12.0-incubating reef-project-0.13.0-incubating reef-project-0.14.0 reef-project-0.15.0

Pom.xml root file

pom.xml

Figure 4: The required details to initialize the prediction and diagnosis tasks for project Apache REEF.

Obtaining a Training Set To learn a fault prediction model we require a training set. In our case, a training set is a set of software components and a labeling that indicates which components are faulty and which are not. Manually labeling the root cause of past bugs is not a scalable solution to obtain a training set. Instead, DeBGUer extracts the training set automatically from the project's *issue tracking* and *version control system*, as described below.

Most projects these days use an issue tracking system, such as Jira and Bugzilla, and a version control system, such as Git and Mercurial. Issue tracking systems record all reported bugs and track changes in their status. They associate each bug with a unique issue ID. Version control systems, track modifications – commits – done to the source files. It is accepted that a commit should contain only the changes that required to address a specific task. A best practice in software development, which is usually enforced, is for the developer to add a message to each commit describing what was done. In particular, commits that fix a bug should write the issue ID in the commit message. DeBGUer uses this information to match fixed bugs to the commit done to fix them. For a bug X, let $\Phi(X)$ denote the set of software components modified in the commit done to fix X. In the absence of manual labeling of faulty software components, DeBGUer assumes that at least one of the in $\Phi(X)$ has caused the bug X. To concretely decide which of these components were faulty, our tool enables two alternative assumptions. The first, which we simply call "all modified", assumes that all components in $\Phi(X)$ have caused the bug. The second, dubbed "most modified", assumes that the component in $\Phi(X)$ that caused the bug is the component whose revisions were most extensive. This is measured by counting the number of lines of code in the components that were either modified, added, or deleted. We chose these two methods due to their simplicity, but note there are more elaborate heuristic methods for identifying the root cause of an observed software bug.

viewed in package `learner.wekaMethods.features` in our source code, which is publicly available at https://github.com/BGU-AiDnD/Debugger.

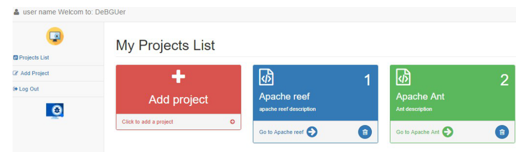

Figure 5: The DeBGUer user home page.

In addition, there are multiple options to define the basic software unit, e.g. class, method or code block. Our tool currently supports defining the based unit as either a class or a method. Thus, the tool enables four configurations: (1) all modified, file-level components, (2) most modified, file-level components, (3) all modified, method-level components, and (4) most modified, method-level components.

To illustrate this automatic labeling process, Figure 2 shows a screenshot of bug Tika-1222 reported for the Apache Tika project using the Jira issue tracker. Figure 3 shows the commit that contains the fix for this bug in the Apache Tika Git repository. Note that the fixed bug id (TIKA-1222) is mentioned in the commit message. Therefore, DeBGUer marks the commit as the ones that fixed the bug. Also, this screenshot shows some of the details recorded by the version control system about the source code modifications made to fix bug. In particular, it shows that this commit changed 2 source files. Hence, DeBGUer will extract the source files or methods changed in the commit. Under the "most modified" assumption and a file-level component, the faulty component of bug Tika-1222 shown in figure 3 is XMLReaderUtils.

Here you can see fault prediciton results of your software code.

Fault prediction results

Search file

file name

#	File name	Access
16	prediction_All_modified_classes.csv	Get file Watch
18	prediction_All_modified_methods.csv	Get file Watch
20	prediction_Most_modified_classes.csv	Get file Watch
24	prediction_Most_modified_methods.csv	Get file Watch

Figure 6: DeBGUer's fault prediction webpage results.

Diagnoser

Next, we describe the diagnosis algorithm used by the DeBGUer diagnoser component, and in particular how it integrates with the fault prediction model described in the previous section. The input to the diagnoser is the observed system behavior in the form of a set of tests that were executed and their outcome – pass or fail. The output is a set of more explanations, where an explanation is a sets of software components (e.g., class or method) that, if faulty, explain the observed failed and passes tests.

The diagnoser we implemented is an extension of the Barinel software diagnosis algorithm (Abreu, Zoeteweij, and

Figure 7: DeBGUer's fault likelihood viewer.

van Gemund 2009; 2011; Hofer, Wotawa, and Abreu 2012; Campos et al. 2013). We provide here a brief description of Barinel. Barinel is a combination of model-based diagnosis (MBD) and spectrum-based fault localization (SFL). In MBD, we are given 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. A diagnosis is a set of non healthy components. The output of an MBD algorithm is a set of *diagnoses*.

In software diagnosis $COMPS$ is a set of software components (classes or methods). Observations (OBS) are observed executions of tests. Every observed test is labeled as "passed" or "failed". This labeling is done manually by the tester or automatically in case of automated tests (e.g., failed assertions).

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; 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). A challenge in applying MBD to software is that a system description – *SD* – is not likely to exist in software. Instead, Barinel considers the *traces* of the observed tests. A *trace* of a test is the sequence of components involved in running it. Traces of tests can be collected in practice with common software profilers (e.g., Java's JVMTI). If a test failed then we can infer that at least one of the components in its trace is faulty. Thus, the trace of a failed test is a conflict, and Barinel considers it as such when computing diagnoses. Then, it uses a fast hitting set algorithm called STACATTO (Abreu and van Gemund 2009) to find hitting sets of these conflicts, which are then outputted as diagnoses.

Barinel computes a *score* for every diagnosis it returns, estimating the likelihood that it is true. The exact details of how this score is computed is given by Abreu et al. (Abreu, Zoeteweij, and van Gemund 2009; 2011). For the purpose of DeBGUer, 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. Previous work using Barinel has set these priors uniformly to all components. DeBGUer uses the output of its fault predictor to provide these priors, as follows. The software fault predictor is a classifier, accepting as input a software component and outputting a binary prediction: is the component predicted to be faulty or not and a confidence score, indicating the model's confidence about the classified class. Let *conf*(C) denote this confidence for component C. We use *conf*(C) for Barinel's prior if C is classified as faulty, and $1 -$ *conf*(C) otherwise.

Architecture

DeBGUer is constructed of several components in a client-server framework. The server side is implemented in PHP and runs a PHP container, and the client side is implemented using AngularJS. The fault prediction component stores the data for each user project version in a SQLITE database, and the learning algorithm is run using WEKA. The diagnoser component is implemented using Maven and Java bytecode instrumentation.

DeBGUer interfaces with version control systems and issue tracking systems in order to extract data about historical versions and bugs. Currently DeBGUer supports the Git version control and Jira and Bugzilla issue tracking systems.

Interface and Use Cases

DeBGUer supports six main use cases: (1) **Adding a new project.** Add a new project to start the LDP on it. (2) **Fault prediction.** Run our fault prediction algorithm over a chosen version of the project. (3) **Automated diagnosis.** Run our diagnosis algorithm over all failed tests of the current version. (4) **Test viewer.** watch all test outcomes of the current version. (5) **Fault likelihood viewer.** watch the fault likelihood for each software component (package, class or

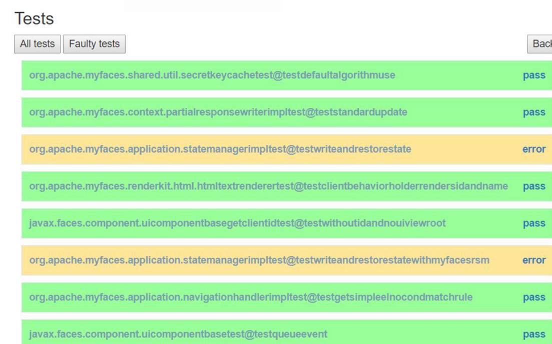

Figure 8: DeBGUer presents the outcomes for each test.

Project name	Start date	#Commits	#Issues	#Files
Ant	2000	14328	6075	1296
CAY	2007	5950	2436	3694
DELTASPIKE	2011	2294	1353	1813
FLINK	2010	14501	10087	6396
IO	2002	2123	571	246
JCLOUDS	2009	10235	1434	5271
KYLIN	2014	7372	3332	1478
MYFACES	2006	4536	3878	1969
OAK	2012	15281	7654	4010
OPENNLP	2010	1832	1209	957
TOBAGO	2004	9663	1901	819
TOMEE	2006	10931	2174	5694

Table 1: Details about the projects we evaluated on.

	All Classes		Most Classes	
Project	Recall	Precision	Recall	Precision
Ant	0.927	0.903	0.958	0.952
CAY	0.995	0.995	0.998	0.997
DELTASPIKE	0.968	0.955	0.981	0.982
FLINK	0.957	0.956	0.982	0.981
IO	0.992	0.984	0.996	0.992
JCLOUDS	0.997	0.994	0.998	0.995
KYLIN	0.919	0.892	0.978	0.968
MYFACES	0.842	0.974	0.974	0.981
OAK	0.963	0.964	0.987	0.974
OPENNLP	0.680	0.463	0.951	0.904
TOBAGO	0.884	0.853	0.957	0.921
TOMEE	0.976	0.969	0.986	0.983

Table 2: Fault prediction results.

method). (6) **Diagnoses viewer.** watch the possible explanations for the observed bug.

Figure 5 shows the DeBGUer's user home page. It is the entry point for the use cases described above. In order to add a new project the user is prompted to provide the Git address of the project, and to add the specific project versions to use in order to build the training set. Then, DeBGUer clones the project and creates the project environment on the DeBGUer server. The project environment also includes the reported bugs from the issue tracker and the code history from the source control.

Once the information from the Git and the issue tracker is collected, DeBGUer starts the fault prediction task. Figure 6 shows a screenshot of the prediction task webpage of DeBGUer. This webpage allows viewing and downloading the prediction files. As explained before, there are four prediction configurations: "all modified classes", "most modified classes", "all modified methods", "most modified methods". The fault likelihood viewer shows the faulty likelihood for each software component. Figure 7 demonstrates the fault prediction viewer webpage. Furthermore, The prediction task outputs CSV files that contains a list of the software components and their probability to contain a fault as predicted by the learning model. Once DeBGUer finishes running the tests, the user can watch the tests outcomes. Figure 8 shows the DeBGUer tests viewer. The viewer shows the outcome for each test in the system.

The prediction model is used by the diagnosis process. This task includes running the system tests and collect their traces and results to isolate the faulty components that caused failing tests. In order to trace the project tests, DeBGUer should build the project and execute the tests. DeBGUer uses the *maven tool* to build the project and the *surefire plugin* to execute the tests. To trace the tests DeBGUer uses *java agent instrumentation*, which is a framework for performing byte-code manipulation in runtime. The output of this task is a list of the diagnosed software components and their probabilities.

Evaluation

We evaluated the fault predictor and diagnoser of DeBGUer on 12 open source projects. All projects were written in Java, use Maven as a building tool, Git as a version control, and are publicly available at https://github.com/apache. Table 1 lists for each of the selected projects, when the project started ("Start date"), the number of commits in Git ("#Commits"), the number of issues in the issues tracker ("#Issues"), and the number of Java source files in the project repository ("#Files"). Unfortunately, we did not have access to an oracle to identify which components are faulty and label them accordingly. Instead, we repeated all our experiments two times: one time using the "most modified" assumption to label the faulty components, and the other time using the "all modified" assumption to label the faulty components. The results for each of these assumptions are presented separately below.

As mentioned above, DeBGUer currently supports two levels of software components: class and method. We conducted experiments on both levels but present here only results for class-level software components, due to space limitations. The results for method-level components are available at github.com/amir9979/aaai19_results.

Fault Prediction

First, we evaluated the fault prediction task. As input we chose four versions from each project as a training set and a later version as a testing set. We evaluate the trained fault prediction models by measuring their precision and recall. In brief, *precision* is the ratio of faulty components among all components identified by the prediction model as faulty. *Recall* is the number of faulty components identified as such by the prediction model divided by the total number of faulty components. Table 2 shows the recall and precision of the fault prediction models for "All Classes" and "Most Classes" configurations. The prediction models generated by Random Forest (with 1,000 trees) for each of the benchmark projects. The results are, in general, impressive, where in all but one project the precision is above 0.9, the recall is more than 0.68 and usually much higher.

Diagnosis results

Next, we evaluated the diagnosis task. The input to a diagnosis task is a set of tests, their traces, and outcomes. The output is a set of possible explanations to the observed failures (diagnoses), each having a score that estimates its correctness. Recall that the diagnoses and their scores are computed by DeBGUer's diagnoser, which uses a modified version of the Barinel algorithm that considers the output of the fault predictor (Elmishali, Stern, and Kalech 2016). For every project we generated 50 different inputs in the follow-

	All Classes			Most Classes		
	Top_k	Recall	Prec.	Top_k	Recall	Prec.
IO	1.00	0.40	0.94	1.06	0.34	0.95
TOBAGO	1.14	0.49	0.84	1.32	0.46	0.61
KYLIN	1.72	0.30	0.62	2.52	0.40	0.53
OAK	2.16	0.18	0.44	4.92	0.20	0.34
OPENNLP	3.72	0.15	0.38	3.62	0.17	0.27
MYFACES	4.24	0.26	0.42	1.24	0.48	0.69

Table 3: Diagnosis evaluation metrics.

ing manner: First, we chose a bug from the test set used to evaluate the fault predictor. Each bug is associated with the set of faulty software components (following either the most-modified or all-modified assumptions). Then, we considered all the JUnit tests written in the code for the packages that contains the faulty components. Then, we choose 50 tests from this set of tests, and simulated their outcomes (pass/fail) by assuming that if a faulty component is in the trace of a test then that test will fail with probability 0.2. We added this probability of a test to pass even if it passes through a faulty component since failures in software are often intermittent.

We evaluated the output of the diagnoser using the following three metrics: Top_k, average recall, and average precision. To compute the Top_k metric, we assume the diagnoses are inspected in order of their score, and return the number of diagnoses that will be inspected until all faults are found. For example, if Top_k=1, it means that diagnosis ranked highest contains the correct diagnosis, while higher values indicate that more diagnoses will have to be inspected in order to find all faulty components. Thus, lower values of top_k are preferred. This measure estimates the usefulness of the diagnoser's output, as it aims to measure the effort required to fix the system using the diagnoser's output. To compute the average recall and average precision metrics, we computed for every diagnosis returned by the diagnoser its precision and recall. Then, we computed weighted average of these precision and recall values, where the results for every diagnosis are weighted by the score given to that diagnosis. This enables aggregating the precision and recall of all diagnoses while considering their likelihood.

Table 3 shows the Top_k, average recall, and average precision of DeBGUer's diagnoser for the evaluated projects. The results show that for most projects, the Top_k metric was particularly low, ranging between 1 and 4.24. In particular, in the IO project, the average Top_k was approximately 1 (we rounded the decimal point after the second digit), which means an almost perfect diagnostic performance. The precision and in particular recall results, however were less impressive. The difference between the positive Top_k results and the negative precision and recall results suggest that our diagnoser can prioritizes the diagnoses effectively, but is not powerful enough to dismiss incorrect diagnoses or to assign them significantly low scores. Improving the scoring mechanism of software diagnosis algorithms is an active field of research. Nevertheless, we argue that from the perspective of usefulness to a developer, low Top_k results are the most important, since they suggest low debugging effort by the developer when using our tool.

Conclusion

This paper describes the current version of our DeBGUer tool, which is an implementation of the LDP paradigm (Elmishali, Stern, and Kalech 2018) for software fault prediction and automated software diagnosis. DeBGUer was tested on 12 projects so far, showing promising results. It is ready to use and avaliable at DeBGUer.ise.bgu.ac.il. The purpose of DeBGUer is to support the dissemination of AI techniques to the software engineering industry and to the open source community. In addition, DeBGUer allows researchers to compare their prediction and diagnosis algorithms to our results, as well as build on our results to develop more sophisticated tools. There are several exciting directions for future work. First, we are currently testing DeBGUer on a significantly larger set of projects. Second, we have not implemented in DeBGUer the test planning aspect of LDP. Third, we intend to perform a user study with actual development teams to demonstrate the benefits of using DeBGUer in the software engineering process.

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.; 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.; Riboira, A.; Perez, A.; and Abreu, R. 2012. Gzoltar: An eclipse plug-in for testing and debugging. In *IEEE/ACM International Conference on Automated Software Engineering*, ASE, 378–381.

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

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

Elmishali, A.; Stern, R.; and Kalech, M. 2016. Data-augmented software diagnosis. In *AAAI*, 4003–4009.

Elmishali, A.; Stern, R.; and Kalech, M. 2018. An artificial intelligence paradigm for troubleshooting software bugs. *Engineering Applications of Artificial Intelligence* 69:147–156.

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

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.

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.

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