AI for Software Quality Assurance Blue Sky Ideas Talk

Meir Kalech, Roni Stern
Ben-Gurion University of the Negev, Israel, {kalech, sternron}@bgu.ac.il

Abstract

Modern software systems are highly complex and often have multiple dependencies on external parts such as other processes or services. This poses new challenges and exacerbate existing challenges in different aspects of software Quality Assurance (QA) including testing, debugging and repair. The goal of this talk is to present a novel AI paradigm for software QA (AI4QA). A **quality assessment** AI agent uses machine-learning techniques to predict where coding errors are likely to occur. Then a **test generation** AI agent considers the error predictions to direct automated test generation. Then a **test execution** AI agent executes tests, that are passed to the **root-cause analysis** AI agent, which applies automatic debugging algorithms. The candidate root causes are passed to a **code repair** AI agent that tries to create a patch for correcting the isolated error.

1 Scientific Background

Testing, debugging, and other software Quality Assurance (QA) activities are performed by software development teams to increase software quality throughout the software development process and in particular prior to deployment. Estimates of the time spent for QA in software products range from 20% to 50% of overall development costs (Tassey 2002). Recent years have demonstrated that the relevant time-to-market for software products decreases rapidly, while the value expected by end-users is not decreasing, thereby leaving less time for QA. This suggests software products quality is expected to degrade, potentially leading to a new software crisis (Cerpa and Verner 2009).

To avoid this crisis, we believe that a novel Artificial Intelligence (AI) paradigm for software development and QA is required. In this paradigm a range of techniques from the AI literature could guide human and computer QA efforts in a cost-effective manner. Figure 1 illustrates the workflow of this AI-driven QA paradigm (AI4QA). Each one of the workflow components, described below, poses an interesting research challenge which can be addressed by different well-known AI methods.

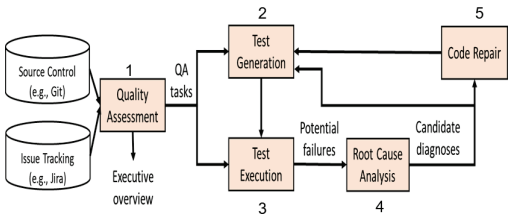

Figure 1: An illustration of the proposed AI4QA paradigm.

A **quality assessment** AI agent (marked 1 in Figure 1) continuously monitors the software development process by collecting information from existing software engineering tools, learning over time to predict where coding errors are likely to occur. A **test generation** AI agent (2) considers the error predictions to direct automated and manual test generation. The **test execution** AI agent (3) executes tests and analyses their outputs, searching for failed tests and abnormal behaviour. These potential failures are passed to the **root-cause analysis** AI agent (4), which applies automatic diagnosis and debugging algorithms to isolate their root cause. In some cases, additional testing is needed to isolate the root cause, in which case the root-cause analysis AI agent will interact with the test generation AI agent to create customized tests to provide additional diagnostic information. After isolation, the candidate diagnoses – root causes – are passed to a **code repair** AI agent (5) that tries to create a patch for correcting the isolated error, or assign it automatically to a human developer that is most likely to be able to solve it. After an error is corrected, the test generation AI agent generates suitable tests to prevent regression. This QA process is constantly monitored by the quality assessment AI agent, which provides immediate feedback to management, suggesting when a software is deployment-ready and which processes need executive attention.

Algorithms for some elements of the AI agents in AI4QA exist in the academic literature and even in industry. However, their adoption has been limited. This is because current algorithms are not powerful enough and do not provide a full solution to the QA process. For example, there are several automated test generation tools (Fraser and Arcuri 2011;

Fraser and Rojas 2019). but without assessing the quality of the tested software or analysing which parts are more likely to contain bugs, it is not possible to guide the test generation in an informed manner.

Indeed, to achieve a ground breaking impact on software development it is crucial not only to develop the individual AI agents in AI4QA, but also to create multi-agent mechanisms that allow these agents to cooperate effectively with each other and with their human counterparts. AI4QA paradigm poses many research and engineering challenges, but the expected outcome is a revolution in how software is developed, leading to increased quality, lower costs, and faster QA process.

2 The State-of-the-Art and Open Challenges

The AI agents in AI4QA paradigm build on range of existing lines of research. We provide a brief summary of these lines of research, highlighting current successes and open challenges to the AI community.

2.1 Software Fault Prediction

Software fault prediction is a classification problem: given a software component – a class, a method, or a line of code – the goal is to determine whether it contains a fault or not. This topic has been studied extensively in the literature, including several recent surveys (Malhotra 2015; 2018). The state of the art approach for building a software fault prediction model is to use supervised machine learning algorithms as follows. As input, these prediction algorithms accept a training set comprising software components and their correct label – faulty or not. Then, they extract various features from every component in the training set, and learn a relation between a component's features and its label. The output of these algorithms is a fault prediction model based on this relation: it accepts features of a software components and outputs a predicted label. Software fault prediction algorithms vary in where they obtain their training set, in the machine learning techniques they use, and the features they extract from each component.

Obtaining a training set. The common approach to obtain a training set is to collect it from issue tracking and version control systems (e.g., Jira and Git). Previously reported bugs are matched with source code revisions that fixed them, and every such pair is a component labelled as faulty in the training set, while all other components are assumed to be correct. Within-project software defect predictions learn a prediction model for a project by using a training set collected from that project (Pan et al. 2019).

Learning algorithms. A range of machine learning algorithms have been used for software defect prediction, such as Random Forest (Elmishali, Stern, and Kalech 2016), Support Vector Machines (Niu et al. 2018), and deep learning (Fan et al. 2019).

Defect prediction features. The features used by existing software prediction algorithms can be categorized into three metric families: code complexity, object-oriented, and process. Code complexity and object oriented metrics are extracted by static code analysis. Code complexity metrics include simple metrics like number of lines of code and more complex metrics such as McCabe's cyclomatic complexity metric (McCabe 1976) and Halsted's metrics (Halstead 1977). Object-oriented metrics measure various object-oriented properties such as the number of classes it is coupled with, and the cohesion of the functions in a class. Process metrics are extracted from the version control system, and try to capture the dynamics of the software development process, such as the number of lines added in the last revision and time since the last edit (Choudhary et al. 2018).

Open Challenges: The accuracy of state-of-the-art prediction algorithms is pretty high. Theoretical models of cost-effectiveness also suggest that guiding QA efforts using software defect prediction should yield high return on investment (Hryszko and Madeyski 2017). However, there are still open challenges:

Imbalance data. There are much more valid software components than buggy components. This is a known challenge for prediction models. The high accuracy reported in the literature is mainly due to the extremely high number of valid components (true negatives), however, the recall is usually still low. Recently prior work explored using techniques for handling imbalanced datasets (Khuat and Le 2019).

Cross-projects learning. Most previous work focus on learning prediction models within project. This approach is not expected to work well for new projects. It is challenging to learn prediction models from multiple projects and use them to generate a prediction model for a new project. This task raises interesting challenges such as how to integrate the learnt prediction models for the new project? How to compute the similarity of the new project to the projects that used for the training? (Goel and Gupta 2020) It seems that transfer learning (Weiss, Khoshgoftaar, and Wang 2016) can be a key to cope with these challenges.

The synergy of software defect prediction in AI4QA. This AI agent is the main driving forces in AI4QA, directing the tasks exerted by all the other AI agents and providing them with a better understanding of the tested software. In addition, input from the other AI agents can further improve existing fault prediction algorithm, e.g., with knowledge about which components have been tested recently.

2.2 Automated Test Generation and Execution

Automatic test generation is one of the most studied problem in automated software engineering (Fraser and Arcuri 2011; Fraser and Rojas 2019). Broadly speaking, a test is an activity that stimulates a system and observes its behaviour. The main challenges addressed in the literature are how a test should stimulate the system under test (SUT), and how to decide whether the observed behaviour is correct or not, corresponding to the test passing or failing, respectively. The first challenge is known as the reliable test set problem and the second challenge is known as the oracle problem (Chen et al. 2018).

Reliable test set problem. Several approaches have been proposed for the reliable test set problem. Random testing is such an approach, in which the SUT is used in different ways

by generating random input values or random sequences of instructions. A different approach for addressing the reliable test set problem is based on symbolic execution (Trabish et al. 2018). Symbolic execution refers to the execution of program with symbols as argument, rather than concrete values. Assignment statements are represented as functions of their (symbolic) arguments, while conditional statements are expressed as constraints on symbolic values. Then, a constraint solver can be used to find input values for the SUT that will execute any desired paths in the SUT. Symbolic execution is also often used in conjunction with real execution of the SUT. This is known as dynamic symbolic execution or concolic testing, and is used in test generation algorithms (Baldoni et al. 2018).

Search-based test generation algorithms approach the reliable test set problem as a combinatorial optimization problem and apply combinatorial search algorithms. To do so, an optimization criteria such as path coverage is defined. Then, search-based test generation algorithms use search algorithms to search the space of possible tests so that the resulting test set maximizes the chosen optimization criteria. EvoSuite (Fraser and Arcuri 2011) is a state-of-the-art example of search-based test generation algorithm that uses a genetic algorithm for searching the space of possible test suites. Such approach has also been used to generate system-level test (Arcuri 2018).

The Oracle problem. The Oracle problem has received less attention than the reliable test set problem, but nonetheless there are several approaches for addressing it (Barr et al. 2014). One approach is to assume that a test fails if it causes the SUT to crash or throw an exception. A second approach is designed for building regression tests: it assumes the current behavior of the SUT is correct, and creates tests to verify that this behavior is maintained. A third approach is to accept from the user some specification or invariants that the SUT must conform to, and verify that they are maintained. For example, recent work by Pill and Wotawa showed how to extract test oracles from specifications given in linear temporal logic (Pill and Wotawa 2018).

Open Challenges: Reliable tests. Despite the many invested efforts to solve this challenge, the industry has not yet adopted these methods and in fact we are far from a working tool. In the light of recent successes of machine learning and deep learning methods in other areas, it could be that these approaches will work also to generate reliable tests.

Oracle. The major limitation of previous approaches that they cope with the simple cases of crash, exceptions and specific user-defined specification rules, however, most tests check the specification and logical errors. Automated oracle for such cases is still challenging.

The synergy of automated testing in AI4QA. An important challenge for the AI4QA paradigm is how to use input from the quality assessment AI agent,to divert testing efforts to areas in the software that are more likely to contain bugs. Moreover, how the test generation AI agent can identify which parts of the software it cannot test properly and require a human tester. Lastly, how the test generation AI agent could be integrated in AI4QA to generate tests on-the-fly during debugging, in order to provide useful diagnostic information.

2.3 Automated Software Diagnosis

Failed tests suggest that one or more software components are not behaving as expected. Software diagnosis, also known as software fault localization and debugging, is the process of finding these faulty components, given the set of passed and failed tests. Various approaches have been proposed for automated software diagnosis (Wong et al. 2016).

Model-based software diagnosis. A logical model of the diagnosed software is assumed or learned, and then used to analyse the discrepancies between this model and the abnormal observations (Pill and Wotawa 2018). This approach is principled and allow using model-based diagnosis algorithms, but cannot scale to real-size programs due to the difficulty of obtaining a useful model for the SUT.

Spectrum-based fault localization (SFL). SFL-based software diagnosis identifies faulty components by collecting traces of executed tests, which are vectors that state which software component was involved in each test. Barinel (Abreu, Zoeteweij, and van Gemund 2011) is a prime example of a software diagnosis algorithm from this approach. It uses a hitting set algorithm to find sets of components such that each set contains a component from the trace of every failed test. Then, it uses an optimization algorithm to rank these sets of components in order to identify the most likely root cause. Many improvements have been proposed to Barinel, such as improving it with information about the system's state (Perez and Abreu 2018).

Slicing-based software diagnosis. Slicing-based software diagnosis removes parts of the SUT that are not relevant to the failed test, thereby narrowing the number of components that may have caused the bug. The challenge in slicing techniques is how to identify and cut out only the relevant parts of the SUT. Some slicing techniques are based on a static analysis of the source code (Kusumoto et al. 2002), while others use dynamic slicing, which analyses a specific execution of the program (Wong et al. 2016). Software diagnosis via program slicing is limited to the size of the smallest slice, which theoretically can be as large as the entire program. However, program slicing can be used effectively in conjunction with another software diagnosis method, e.g., by first narrowing down to the smallest slice that maintain the desired behaviour, and then applying a different approach on that slice (Hofer and Wotawa 2012).

Text-based software diagnosis. A fundamentally different approach that has gain some attention in recent years is to learn the relation between the text in the bug report – the report in which the bug is described – and the location of its root cause. These techniques consider the source code of the SUT as a collection of text document, and apply techniques from the Information Retrieval literature to search for software components that contain similar text to the text in the bug report (Khatiwada, Tushev, and Mahmoud 2018). The main advantage of these approaches is that they do not require executing the SUT, which is time consuming and can

be challenging due to configurations differences. However, these methods rely on having meaningful bug reports with text that suggests the location of the bug.

Open Challenges: Although the topic of software diagnosis has been studied in recent years and there are even tools for this task (Elmishali, Stern, and Kalech 2019), these algorithms are not commonly used in industry, probably the next challenges should be addressed first.

Isolating difficult bugs. Current automated debugging methods succeed to isolate bugs that mostly would be easily isolated by human. There is still a challenge to isolate bugs that are known to be difficult to isolate such as bugs of omission, which are bugs due to missing lines of code, or bugs that caused due to multi-thread programming.

Diagnoses ranking. Software diagnosis algorithms tend to produce a large number of candidate diagnoses, thus ranking the diagnoses effectively is crucial and challenging.

The synergy of root cause analysis in AI4QA. In the proposed AI4QA paradigm, an important challenge of the root-cause analysis AI agent is to identify the suitable course of action for isolating each bug. This includes challenges as how to choose the appropriate diagnosis algorithm for the observed failure, and how to identify when delegating to a human is a better option.

2.4 Automated Software Repair

The next step after diagnosing the root cause of a bug is to fix it. There are several algorithms that attempt to automate the bug fixing step. The output of these algorithms is a patch that fixes the bug, which is then presented to the human developer to approve or reject. Some software repair algorithms are general-purpose while others focus on specific types of bugs. For example, Code Phage is a code repair algorithm designed specifically for repairing buffer overflow bugs (Sidiroglou-Douskos et al. 2015). Some algorithms restrict themselves to use specific code repair templates, or try to learn from previously used patches (Long and Rinard 2016), while others try to synthesize the patch by adding and manipulating raw code instructions. A recent comprehensive survey (Gazzola, Micucci, and Mariani 2017) divided software repair techniques to two classes: generate-and-validate and semantic driven.

Generate-and-validate software repair. Software repair algorithms from this class modify the existing code in various ways until all tests pass, without a deep understanding of what the code is supposed to do. To this end, several atomic code manipulation operators are defined and the software repair problem becomes a search problem of finding the sequence of code manipulation operators that results in passing tests. Since the number of possible sequences of operators is infinite, software repair algorithms apply a range of heuristic search algorithms to search this space effectively. GenProg (Le Goues et al. 2011) uses a genetic algorithm to search for the appropriate sequence of code manipulations. Then, it applied delta debugging to reduce found sequence code manipulation so that as to produce a minimal patch. Other code repair algorithms from this class use different search algorithms, different code manipulation operators, and different fault localization methods.

Semantic-driven software repair. Algorithms that follow this approach convert the software repair problem to a software synthesis problem. They usually have three main steps: behavior analysis, problem generation, and fix generation. In the behavior analysis step, the system's incorrect behavior (the bug) and the system's desired behavior are both encoded in some formal way. In the problem generation step, a formal problem is defined such that its solution is a patch that corrects the system's behavior. In the fix generation step, the generated problem is solved, e.g., using a general problem solving technique, outputting the desired patch. For example, the Angelix algorithm reduces the system repair problem to a Partial Max Satisfiability-Modulo-Theory (pMaxSMT) problem, and then applies a general-purpose pMaxSMT solver. For the behavior analysis step, Angelix relies on the software tests to define the desired behavior. An alternative is to derive the desired behavior for software specifications or from a reference implementation (Mechtaev et al. 2018).

Open Challenges: Current software repair algorithms work surprisingly well on academic benchmarks. However, this has not translated to the expected transformation in how software is developed in industry.

Efficiency. Existing algorithms are not powerful enough. The search space of repair is extremely high even for discrete variables and thus a major challenge is to focus the search on more probable repair patches. This can be investigated by learning methods.

Isolation and repair tradeoff. As mentioned, the root-cause analysis AI agent returns a large set of candidate diagnoses, where most of them are false positive. Repairing false positive candidate bugs is time consuming. There is a tradeoff between the bug isolation process and the bug repair. The more efforts invested to isolate the correct bug the less required efforts in repairing it. Balancing between these two tasks is challenging.

The synergy of repair in AI4QA. Fundamental questions that must be answered is when an identified bug can be fixed by an automated repair algorithm and when it is better to delegate it to a human developer. To properly address this question, a preliminary challenge should be addressed, how to assess the importance of an identified bug and its predicted complexity to repair. Given the broader view of the developed project, the code repair AI agent will choose the appropriate course of action for each bug: whether to delegate or solve by a code repair algorithm, and if so, which code repair algorithm to use.

3 From Blue-Sky to Practical Research

Developing successful and efficient AI algorithms to address the presented open challenges will make a revolution in how QA is maintained during software development. AI agents will collaborate effectively with each other and with their human counterparts in a broad range of QA activities. Instead of investing human efforts in each one of the QA components: testing, debugging and repair, adopting the AI4QA

paradigm, proposed in this paper, will reduce the human efforts and accelerate the QA process.

Acknowledgement

We would like to thank the Israel National Cyber Bureau, for supporting this research.

References

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.

Arcuri, A. 2018. Evomaster: Evolutionary multi-context automated system test generation. In *ICST*, 394–397. IEEE.

Baldoni, R.; Coppa, E.; D'elia, D. C.; Demetrescu, C.; and Finocchi, I. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51(3):50.

Barr, E. T.; Harman, M.; McMinn, P.; Shahbaz, M.; and Yoo, S. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41(5):507–525.

Cerpa, N., and Verner, J. M. 2009. Why did your project fail? *Communications of the ACM* 52(12):130–134.

Chen, T. Y.; Kuo, F.-C.; Liu, H.; Poon, P.-L.; Towey, D.; Tse, T.; and Zhou, Z. Q. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51(1):4.

Choudhary, G. R.; Kumar, S.; Kumar, K.; Mishra, A.; and Catal, C. 2018. Empirical analysis of change metrics for software fault prediction. *Computers & Electrical Engineering* 67:15–24.

Elmishali, A.; Stern, R.; and Kalech, M. 2016. Data-augmented software diagnosis. In *IAAI/AAAI Conference*.

Elmishali, A.; Stern, R.; and Kalech, M. 2019. Debguer: A tool for bug prediction and diagnosis. In *Proceedings of AAAI*, volume 33, 9446–9451.

Fan, G.; Diao, X.; Yu, H.; Yang, K.; and Chen, L. 2019. Software defect prediction via attention-based recurrent neural network. *Scientific Programming* 2019.

Fraser, G., and Arcuri, A. 2011. Evosuite: automatic test suite generation for object-oriented software. In *ACM SIGSOFT symposium and the European conference on Foundations of software engineering*, 416–419.

Fraser, G., and Rojas, J. M. 2019. *Software Testing*. Cham: Springer International Publishing. 123–192.

Gazzola, L.; Micucci, D.; and Mariani, L. 2017. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45(1):34–67.

Goel, L., and Gupta, S. 2020. Cross projects defect prediction modeling. In *Data Visualization and Knowledge Engineering*. Springer. 1–21.

Halstead, M. H. 1977. *Elements of Software Science (Operating and Programming Systems Series)*.

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

Hryszko, J., and Madeyski, L. 2017. Assessment of the software defect prediction cost effectiveness in an industrial project. In *Software Engineering: Challenges and Solutions*. Springer. 77–90.

Khatiwada, S.; Tushev, M.; and Mahmoud, A. 2018. Just enough semantics: An information theoretic approach for ir-based software bug localization. *Information and Software Technology* 93:45–57.

Khuat, T. T., and Le, M. H. 2019. Ensemble learning for software fault prediction problem with imbalanced data. *International Journal of Electrical & Computer Engineering (2088-8708)* 9.

Kusumoto, S.; Nishimatsu, A.; Nishie, K.; and Inoue, K. 2002. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering* 7(1):49–76.

Le Goues, C.; Nguyen, T.; Forrest, S.; and Weimer, W. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38(1):54–72.

Long, F., and Rinard, M. 2016. Automatic patch generation by learning correct code. In *ACM SIGPLAN Notices*, volume 51, 298–312. ACM.

Malhotra, R. 2015. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing* 27:504–518.

Malhotra, R. 2018. An extensive analysis of search-based techniques for predicting defective classes. *Computers & Electrical Engineering* 71:611–626.

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

Mechtaev, S.; Nguyen, M.-D.; Noller, Y.; Grunske, L.; and Roychoudhury, A. 2018. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering*, 129–139.

Niu, Y.; Tian, Z.; Zhang, M.; Cai, X.; and Li, J. 2018. Adaptive two-svm multi-objective cuckoo search algorithm for software defect prediction. *International Journal of Computing Science and Mathematics* 9(6):547–554.

Pan, C.; Lu, M.; Xu, B.; and Gao, H. 2019. An improved cnn model for within-project software defect prediction. *Applied Sciences* 9(10):2138.

Perez, A., and Abreu, R. 2018. A qualitative reasoning approach to spectrum-based fault localization. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 372–373. ACM.

Pill, I., and Wotawa, F. 2018. Automated generation of (f) ltl oracles for testing and debugging. *Journal of Systems and Software* 139:124–141.

Sidiroglou-Douskos, S.; Lahtinen, E.; Long, F.; and Rinard, M. 2015. Automatic error elimination by horizontal code transfer across multiple applications. In *Acm Sigplan Notices*, volume 50, 43–54. ACM.

Tassey, G. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project* 7007(011):429–489.

Trabish, D.; Mattavelli, A.; Rinetzky, N.; and Cadar, C. 2018. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*, 350–360. ACM.

Weiss, K.; Khoshgoftaar, T. M.; and Wang, D. 2016. A survey of transfer learning. *Journal of Big data* 3(1):9.

Wong, W. E.; Gao, R.; Li, Y.; Abreu, R.; and Wotawa, F. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42(8):707–740.