Improving Proceeding Test Case Prioritization with Learning Software Agents

Sebastian Abele and Peter Göhner
Institute of Industrial Automation and Software Engineering, University of Stuttgart, Stuttgart, Germany

Keywords: Machine Learning, Test Case Prioritization, Test Suite Optimization, Software Agents.

Abstract: Test case prioritization is an important technique to improve the planning and management of a system test. The system test itself is an iterative process, which accompanies a software system during its whole life cycle. Usually, a software system is altered and extended continuously. Test case prioritization algorithms find and order the most important test cases to increase the test efficiency in the limited test time. Generally, the knowledge about a system's characteristics grows throughout the development. With better experience and more empirical data, the test case prioritization can be optimized to rise the test efficiency. This article introduces a learning agent-based test case prioritization system, which improves the prioritization automatically by drawing conclusions from actual test results.

1 INTRODUCTION

The development of systems with a high quality is an important factor to succeed in the market. The high competition leads to a decreasing time-to-market. Hence, the development process must be carried out efficiently. One of the major parts of the test process is the system test. The system test is a process which accompanies the whole development process and can't be considered isolated. With about 80%, the largest quantity of the total test expenditure is spent to the regression test (Chittimalli and Harrold, 2009). In the regression test, already available test cases are executed repeatedly to find faults that may have been newly introduced with changes in the system. Over time, the test suites for regression testing may grow to very large repositories with thousands of test cases. Executing all the test cases takes a vast amount of time and resources which are often not available due to the short time-to-market.

The planning of a test run with the selection of appropriate test cases is a very complex time intensive task. A lot of data has to be considered to achieve a very efficient test plan. In order to address these challenges, computer-aided test case selection and prioritization techniques are used to find the most important test cases for the available time slots automatically. Test case selection techniques reduce the test suites by identifying only relevant test cases, for example based on the coverage of changes of the source code since the last test run. An overview over different test selection techniques can be found in (Engström et al., 2010). Prioritization techniques order the test cases by their expected benefit for the test. Unlike test case selection, a test run that is executed on the base of prioritized test cases may be interrupted at any time having still the maximal benefit possible to the interruption time. (Yoo and Harman, 2012) describe test case selection and prioritization techniques, which have been developed in the past decades. The test case prioritization techniques have in common that they calculate the test case order to specific times before a new test run starts.

The knowledge about the tested system grows from test run to test run. More and more data, like fault histories, are collected and evaluated to generate the test case order for the next test run. Not only the collected data, but also the knowledge about the tested system and the developer and tester experience is growing. The tested software may show some unexpected behavior in the test, which is understood better and better by the test engineers. The classic test case prioritization techniques are usually not adapted to grown knowledge and experience. The test case prioritization may not be optimal with respect to the new knowledge.

To improve the test case prioritization with the grown knowledge during the test process, we propose a test case prioritization system, which uses machine learning approaches. With machine learning, the test case prioritization algorithm is updated every test run with the actual test results. The test case prioritization algorithm and the machine learning approaches are developed using software agents. Chapter 2 describes the agent-based test case prioritization and the used prioritization algorithm. This approach is extended by a machine learning approaches in chapter 3. Finally, chapters 4 and 5 describe further research plans and ideas.

2 AGENT-BASED TEST CASE PRIORITIZATION

The paradigm of agent-based software development is well suited to conquer the issues given by the boundary conditions in a system test. Agent systems are software systems, which consist of various agents. An agent is a piece of software, which is capable to act largely independent to fulfill the given goals. In an agent system, different agents cooperate to achieve a superior goal. Agent systems and their development are described in (Wooldridge and Jennings, 1995) and (Mubarak, 2008).

In the field of prioritizing test cases, the agents collect and integrate information about the tested system. Using this information they provide a list of prioritized test cases. An agent-based test case prioritization system has been developed by (Malz and Göhner, 2011). Each software module and each test case is represented by one agent. The prioritization process consists of two main steps: First the test module agents predict the fault-proneness of the software modules for the next test run. In the second step, the test case agents calculate the fault-revealing probability of the test cases. For every module, the test case agents calculates how probable it is that the represented test case reveals faults in that module. By calculating the mean value of all fault-revealing probabilities weighted with the fault-pronenesses of the modules, the priority value is obtained.

The fault-proneness is calculated by evaluating different metrics for each software module. Fault-proneness prediction is described for example in (Kim et al., 2007). (Bellini et al., 2005) compare different models to estimate the fault-proneness. The metrics are differentiated in white box and black box metrics. White box metrics base on the analysis of the source code, e.g. to cover changes since the last test run. Especially when the software system is developed by different departments or even different companies together, the access to the source code may be restricted and the white-box metrics not available. Black-box metrics represent information about the modules, which is available without direct access to the source code. Part of the black-box metrics are fault and change histories and developer-given criticality and complexity values.

This kind of agent-based systems is predestined to run in a distributed environment. Information about a software system that is under test is usually distributed in such an environment. Every department or company holds information about the part of the software, which is developed by the department or by the company. Agent-based test case prioritization allows to integrate the distributed information to generate the test case order for the system test. By operating the relevant agents locally, departments or companies keep the control over their data. The agents only deliver data, which is necessary for the current prioritization task. A concept for an agent-based information retrieval system was described by (Pech and Goehner, 2010).

The agent-based test case prioritization system uses fuzzy-logic rules, which reflect expert knowledge about the relation of the metrics to the fault-proneness and the fault-revealing probability. A description of the fuzzy logic rules for test case prioritization can be found in (Malz et al., 2012). The rules mainly state common relations like "a fault-prone module in the past will be fault-prone in the future", "complex modules are more fault prone than simple ones" or "test cases, which found a lot of faults in the past, will find a lot of faults in the future". Like (Fenton and Neil, 1999) states, expert knowledge often is only an expert opinion, when the fault-proneness is predicted by them. This statement is applicable to the fuzzy-logic rules on the one hand but also to some evaluated parameters like complexity and criticality values, which are obtained by the developers. With additional knowledge about the system, e.g. with the actually found faults during the test, the fault-proneness prediction can be evaluated and improved. Therefore learning mechanisms are integrated into the agents.

3 IMPROVING PRIORITIZATION BY LEARNING AGENTS

In the classic agent-based test case prioritization approach, the prioritization is determined using fuzzy-logic. Rules are formalized, which reflect common expert knowledge, which is usually valid for all software development projects. In reality, the software development projects may have deviations because the size, purpose, development team and many other factors differ from project to project. Adapting the prioritization algorithm manually to the project specifics is nearly impossible because the specifics are usually unknown and hard to be identified. With the usage of learning algorithm, the prioritization system is able to adapt itself using the knowledge from actual performed tests.

3.1 Learning Fault-proneness Prediction

As a first step towards a learning agent-based test case prioritization system, the prediction of the fault-proneness has been extended by a genetic algorithm. The usage of the genetic algorithm is depicted in figure 1. The classic approach uses fuzzy logic rules to estimate the fault-proneness out of the parameters also shown in the figure. The calculated fault-proneness value is compared with the number of actually found faults in the test. The difference between the predicted fault-proneness and the actually found faults is given to the genetic algorithm as a correction value for the next test run.

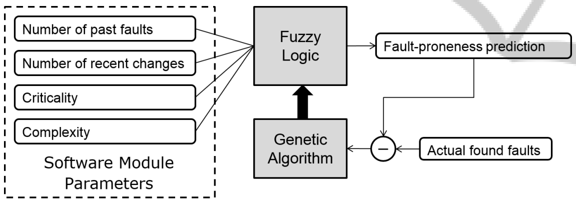

Figure 1: Usage of a Genetic Algorithm to improve the fault-proneness prediction.

The genetic algorithm optimizes the fuzzy logic rules in that way that the fault-proneness prediction would have given a value, which was much closer to the number of actually found faults. For the next test run, the fuzzy logic will use the optimized rule set and provide a better prediction of the fault-proneness value. With the better fault-proneness a better test case prioritization is achieved.

The genetic algorithm has been applied to optimize the weight of the single rules. The rule weight reflects the influence of a parameter to the fault-proneness. The more the fault-proneness depends on a single parameter, the higher the weight of the rules representing this parameter is. The weight of all rules is combined to a *chromosome* for the genetic algorithm. With inheritance and random mutation, the genetic algorithm searches for a new, evolved chromosome, which is better suited to calculate the fault-proneness. Therefore it creates a various number of chromosomes and compares their *fitness*. The fitness is determined by comparing the prediction result of the fuzzy logic rules using the current chromosome as rule weights with the number of actually found faults. The smaller the difference, the higher the fitness.

Since not every rule is used necessarily for each fault-proneness calculation, only the weights of rules that are used should be changed to avoid unwanted side effects. If a mutation to a weight doesn't have an influence to the result, then this weight is locked and not changed anymore. The chromosome with the best fitness is given to the fuzzy logic. The fuzzy logic adapts the weight to it's rules for the next fault-proneness prediction.

3.2 Evaluation of the Learning Fault-proneness Prediction

To evaluate the learning fault-proneness prediction, it is compared against the classic approach that doesn't learn. As an example project, a computer game was chosen that has been developed during a 24 hours programming competition. The boundary conditions of this competition assure that the developer is time-bounded in implementation and test. Additionally, we assume that the development process is fundamentally different to classic software development in a short-time programming competition. Therefore the classic rules may not be optimal and can be optimized using the learning approach.

The game was divided into five modules, which have been investigated: Underground, Obstacle, Player, Enemy and Background. Before the development started, the developer assigned complexity and criticality values to the modules (see Table 1). As third parameter, the relative development effort to the single modules has been recorded as an indicator of the changes done to the modules in the development phases (see Table 3). Additionally, the faults that were revealed were recorded for further analysis (see Table 2). The fault-proneness has been calculated three times during the 24 hours, once at the beginning of the project, once in the middle and once shortly before the deadline.

Table 1: Parameters given by the developer.

	Complexity	Criticality
Underground	10	6
Obstacle	1	5
Player	8	10
Enemy	6	5
Background	7	7

To assess the quality of both fault-proneness prediction methods, a reference value has been calculated. The reference value is assembled by the actual number of revealed faults, the severity of the faults

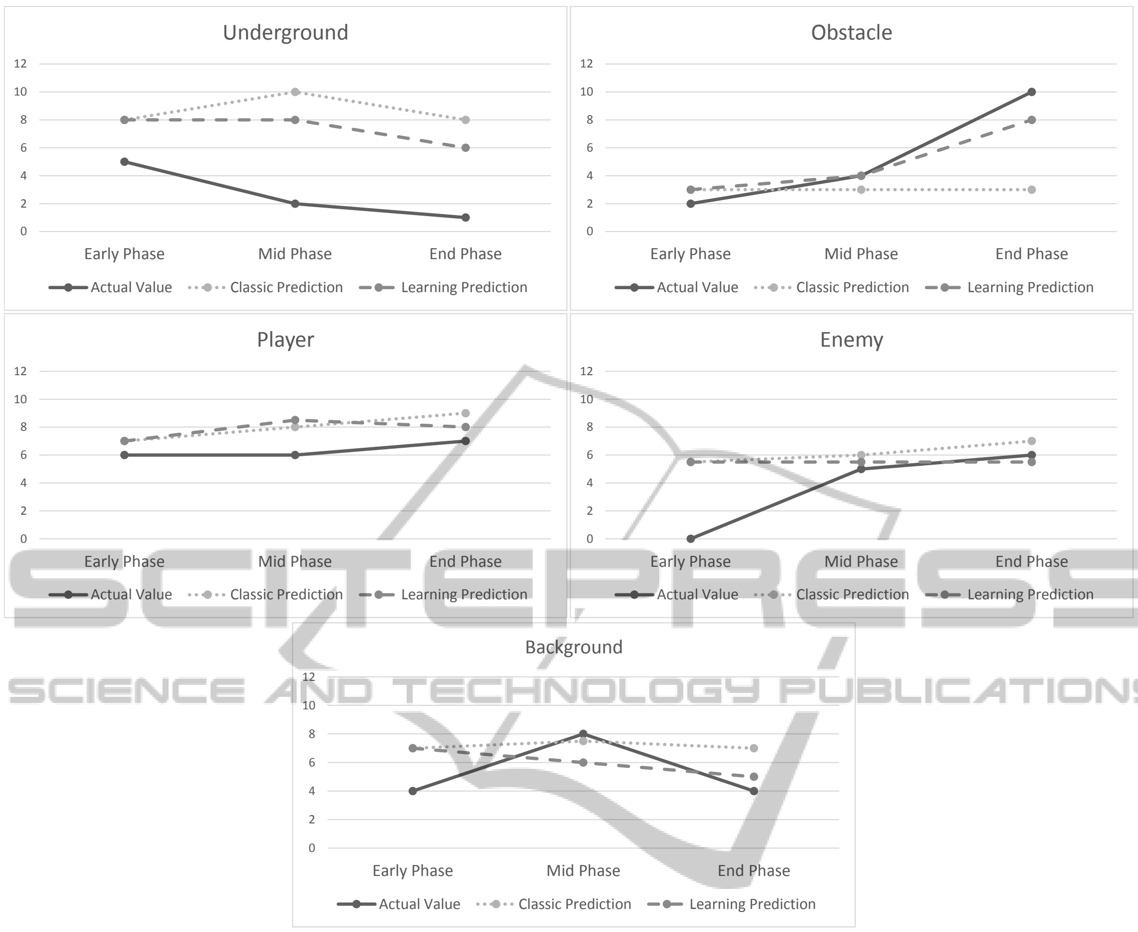

Figure 2: Comparison between classic prediction and learning prediction.

Table 2: Real found faults in early, mid and late phases.

	Early	Mid	Late
Underground	3	7	3
Obstacle	1	3	4
Player	0	2	5
Enemy	0	3	8
Background	1	5	2

Table 3: Spend effort in early, mid and late phase.

	Early	Mid	Late
Underground	49%	12%	6%
Obstacle	13%	10%	31%
Player	27%	36%	22%
Enemy	6%	22%	32%
Background	5%	20%	9%

and a postmortem estimation of the developer, which module would have been the most important to test. Figure 2 shows the result and comparison between the approaches. By analyzing the results and the given scenario the following facts have been witnessed:

- The judgment of the developer for the criticality and complexity values is not fitting very well. He overestimated the criticality of the Underground module, which may lead to an excessively high predicted fault-proneness.
- For the obstacle module the developer underestimated the complexity. In reality that module was more fault-prone than expected.
- The trend of the found faults follows the order in which the modules were developed. The timing of the development has a big impact to the fault-proneness in this scenario.
- For all modules, the prediction, which uses the genetic algorithm, is closer to the actual value than the classic prediction.

Those results show that the scenario in which the software is developed has a large impact on the fault-proneness, and the prediction of it. Especially in such a very short-termed development project, the classic factors like criticality and fault history are quite weak for the fault proneness calculation. However, the results show that the used genetic algorithm is capable to correct the unfavorably selected parameters and weights for the calculation. The impact of the unfortunate chosen criticality values is weakened.

3.3 Critique of the Approach

Despite the results of the study, there are some things that must be considered. The fault-proneness prediction is optimized using the actually found faults in the performed test. This brings two requisites: The test cases, which test the software module, must be good enough to find a representative number of faults. Assessing the quality of test cases is a very complex task itself. Secondly, at least some of the test cases must have been executed in the test run. If the test cases are not executed due to time limitations, then there is no optimization possible. The algorithm should not run with a number of actually found faults, which is too small to be representative.

The currently implemented genetic algorithm is only optimizing the weight factors of the single fuzzy logic rules. There may be other relations, which make it necessary to change the rules themselves, add new rules or change fuzzyfication and defuzzyfication functions. This will be investigated in further research.

4 PLAUSIBILITY CHECKING

Another conclusion of the results presented in chapter 3.2 is that not only the algorithm itself may be in focus for a learning agent system. Other data, like the complexity value, can also be corrected with actual test results. In that case, the wrong complexity would be corrected instead of eliminating its influence to the fault-proneness. The correction of data brings a lot of challenges for further work. It is necessary to distinguish between wrong parameter data and a wrong estimated fault-proneness. The first step towards improving input data of the fault-proneness prediction and test case prioritization is to check the data plausibility. The plausibility can be checked on two levels. Firstly, the consistency of all input data is ensured. Secondly, further conclusions are generated to find faults or uncertainties in the parameters, which are not violating the consistency.

4.1 Consistency Checking

All parameters, which are evaluated to generate the test case prioritization must be consistent in order to draw valid conclusions. For the consistency, rules can be established. A lot of consistency checking is already done by state of the art test management tools. Especially the establishment of invalid dependencies is denied and the traceability is ensured. Nevertheless, the current consistency checking needs to be extended by the incorporation of actual test results and other feedback from the test. When an inconsistency occurs, the test management system should be capable to give the user detailed information about the reason or even to fix the inconsistency automatically. To generate this information, more evaluation needs to be done. One objective of further research is to generate consistency rules and to add further data evaluation to generate conclusions, which help to fix the inconsistency. The following list gives some examples for consistency rules and the conclusions, which can be drawn:

- A test case can only find faults in software modules, which are covered or which are dependent from covered modules. If the test case finds a fault in another module, either the coverage is wrong or a dependency is missing.
- The interrelations between requirements, functionalities and modules must be consistent with the test case coverage. If a fault is revealed in a module, which is not implementing the tested functionality, the dependencies are set wrong.
- Especially when modules have been added or deleted, it must be ensured that all relations and dependencies are updated as well.

(Rauscher and Göhner, 2013) developed an agent-based approach to ensure the consistency of a set of concurrent system models of a mechatronic system. They use an agent and an ontological description for each system model. Based on the ontological representation of the models, they define consistency rules, which must be fulfilled. The rules are verified by the agents automatically when one or more models are changed. The consistency checking approach is adapted and evolved to be used inside the agent-based test management system. Consistency rules will be generated and verified using information from actually performed tests.

4.2 Further Recommendations to Improve the Data

Faulty or poorly chosen parameters, which don't violate plausibility rules, are harder to find. Nevertheless, recommendations can be generated, which indicate that the data should be checked manually. In the evaluation study, the developer misjudged the complexity of several software modules. After applying the genetic algorithm, the weight of the complexity rules was lowered significantly. Instead of lowering the weight, the misjudged values can be corrected. If a lot of faults are found in a module, which has a quite low complexity, this value might be reassessed and changed.

5 SWARM LEARNING

The test case prioritization system consist of a set of individual software agents. Currently, the agents are acting widely independent in order to calculate the fault-proneness or the fault-revealing probability. The implemented learning algorithm improves the fault-proneness calculation for each agent individually. By adding swarm intelligence features to the agents, they will be capable to generate further information collectively. By comparing learning results of a parameter, the agents can distinguish if a learned characteristic is valid in the whole project, in a specific part of the software ore only in a single module. If the characteristic of the single module is different to all others, this is a hint to a possibly wrong parameter value.

Characteristics, which are common in the whole project, can be abstracted and used as basic knowledge for newly introduced modules or test cases. The agents, which represent these modules or test cases don't need to learn the common characteristics and can provide a better result earlier. The comparison of the learning results also helps to identify wrongly learned relations. Strong deviations and fluctuations can be detected and corrected.

6 CONCLUSIONS

In this article, we introduced a learning agent-based test case prioritization system. The system uses a genetic algorithm to improve the test case prioritization with growing knowledge from the proceeding development and test process. Our analysis with a learning fault-proneness calculation as a base for the test case prioritization showed that the learning agents are able to improve the prioritization significantly. Especially if the evaluated information is wrong or imprecise, e.g. because of a misjudgment of the developers in the complexity of a module, the learning algorithm helps to reduce the effect of this inaccurate parameters.

In our future work, we will extend the used genetic algorithm to the fault-revealing calculation of the test cases. In parallel, we investigate further techniques, which may help improving the test case prioritization, for example by the realization of the consistency checking and swarm intelligence described in chapters 4 and 5.

REFERENCES

Bellini, P., Bruno, I., Nesi, P., and Rogai, D. (2005). Comparing fault-proneness estimation models. In *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 205–214.

Chittimalli, P. and Harrold, M.-J. (2009). Recomputing coverage information to assist regression testing. *IEEE Transactions on Software Engineering*, 35(4):452–469.

Engström, E., Runeson, P., and Skoglund, M. (2010). A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30.

Fenton, N. and Neil, M. (1999). A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689.

Kim, S., Zimmermann, T., Whitehead Jr., E. J., and Zeller, A. (2007). Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, pages 489–498, Los Alamitos. IEEE Computer Society.

Malz, C. and Göhner, P. (2011). Agent-based test case prioritization. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 149–152.

Malz, C., Jazdi, N., and Göhner, P. (2012). Prioritization of test cases using software agents and fuzzy logic. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 483–486.

Mubarak, H. (2008). Developing flexible software using agent-oriented software engineering. *IEEE Software*, 25(5):12–15.

Pech, S. and Goehner, P. (2010). Multi-agent information retrieval in heterogeneous industrial automation environments. In *Agents and Data Mining Interaction*, volume 5980 of *Lecture Notes in Computer Science*, pages 27–39. Springer, Berlin and Heidelberg.

Rauscher, M. and Göhner, P. (2013). Agent-based consistency check in early mechatronic design phase. In *Proceedings of the 19th International Conference on Engineering Design (ICED13), Design for Harmonies*, volume 9, pages 289–396. Design Society, Seoul.

Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(02):115–152.

Yoo, S. and Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120.