Reinforcement Learning for Reliability Optimisation

Prasuna Saka
Advanced Systems Laboratory
DRDO, India
Email: prasunas@asl.drdo.in

Ansuman Banerjee
Indian Statistical Insitute
Kolkata, India
Email: ansuman@isical.ac.in

***Abstract*—Software Reliability Optimization problem is aimed at bridging the reliability gap in an optimal way. In an industrial setting, focussed testing at the component level is the most favored solution exercised to fill the reliability gap. However, with the increased complexity in the software systems coupled with limited testing timing constraints finding an optimal set of test suite to bridge the reliability gap has become an area of intense research. Furthermore, the stochastic nature of the reliability improvement estimates associated with each test suite manifolds the complexity. Here, we propose Reinforcement Learning (RL), as a mechanism to find an optimal solution. We have shown how an interactive learning is used to estimate the true outcome of the selection and the action selection policy so as to maximise the long term reward. The estimation methodology and the selection policy is inspired by Multi-armed bandit solution strategies. Firstly, we employ a sample average estimation technique for deriving the true outcomes. Secondly, a variant of simple greedy algorithm coined as epsilon-greedy algorithm is considered for action selection policy. These two steps are iteratively exercised until the selection criteria converges. The efficacy of the proposed approach is illustrated on a real time case study.**

Keywords–Reliability Optimisation; Reinforcement Learning; Multi-armed bandit.

I. INTRODUCTION

To beat the modern systems software complexity, software designs are built hierarchically - from the conceptual architectural model gradually progressing towards leaf-node components/blocks. In parallel, architectural based software reliability analysis [1][2] has gained prominence in recent years to asses the reliability of the overall system. Intuitively, in an hierarchical system, overall reliability of a system improves by improving the reliability of the underlying components. Often it is found that the existing reliability of the system is less than the intended reliability - we call this difference as the *reliability gap*. Ensuring that the reliability gap is closed is of a major concern for mission critical software and therefore has been the forefront of active research over decades. In general, the reliability of the components can be improved in the following ways:

- By using more *focussed test suites* so that more testing will increase its functional reliability.
- By introducing *redundancy* for weak functional parts.

We may select any (or a combination) of the above mentioned approach. However, improving reliability by focussed testing is the preferred option over providing software redundancy for two notable reasons i) Redundant software adds more to the existing complexity ii) Increased foot print size by incorporating software redundancy leads to undesirable loads on the memory needs of the system.

Focussed testing looks for testing a sub-set of the software components so that the reliability is enhanced to the intended level. As each component testing has different contribution to the reliability improvement figure, their exists *multiple non-dominating solutions* to raise the reliability of the overall software to the desired level. Each of these solutions acts as a representative to the problem. Now, the reliability optimization problem tries to minimize the efforts of testing so that the optimized solution maps to any of the non-dominating solutions.

The remainder of the paper is organised as follows. Section 2 emphasises the problem area using a motivating example. Section 3 presents a brief description of the methodology adopted in seeking the solution. Section 4 describes a real case study, and formulates the numericals of the problem. Section 5 presents experimental results considering several indicative scenarios. Section 6 discusses the related work and Section 7 offers concluding remarks.

II. MOTIVATING EXAMPLE AND PROBLEM FORMULATION

We introduce a small instructive example to disclose the intricacies involved with the problem. For illustration purpose, let us consider a software having 6 basic components swc_1, swc_2,.... swc_6 and each component is provided with a test suite tc_1, tc_2,.... tc_6. As each component has its own contribution to the overall software reliability, reliability improvement figures Ri_i*'s* which can be obtained by testing the component with its associated test suite would be different. Column 2 of Table I represents Ri_i*'s*, and column 3 corresponds to test suite execution times. Assume the current reliability R_{curr} as 0.6 and target reliability R_d be 0.95. Now, our objective is to find an optimal set of test suites such that the reliability gap of 0.35 is closed. In general, one can figure out the solution sets informally by treating it as a combinatorial problem and make different combinations of test suites until the reliability gap is filled or one can attempt on a formal note using heuristic search based algorithms. For this simple case, one can informally derive the solution sets using pen and paper. Multiple solution forms to fill the reliability gap are presented in Table II. Among the various solutions sets, the set represented by S2 is the optimal. Note that these solution sets are not complete and hence there is a possibility of having a better selection set than S2. It is worthy to note that, both pen and paper and heuristic search based methods become intractable with the increase in the test suite collection.

Copyright (c) IARIA, 2018. **ISBN: 978-1-61208-668-2**

TABLE I. **AN EXAMPLE SHOWING THE RELIABILITY IMPROVEMENT FIGURES Ri_i's and EXECUTION TIMES OF EACH TEST SUITE.**

Test suite	Ri_i	t_i(man hours)
tc_1	0.05	7
tc_2	0.1	5
tc_3	0.15	5.5
tc_4	0.125	6
tc_5	0.195	9
tc_6	0.23	8

TABLE II. **SOLUTION SETS.**

Id	Solution set	t_i(man hours)
S1	$\{tc_6, tc_5\}$	17
S2	$\{tc_2, tc_3, tc_4\}$	16.5
S3	$\{tc_6, tc_2, tc_3\}$	18.5
S4	$\{tc_1, tc_2, tc_3, tc_4\}$	23.5

Now, we introduce another element which manifolds the problem scenario - the scenario when the *Ri_i's* represented are just *estimates* not *actual values*. In this consequence, it is a natural choice to associate a confidence figure for each estimate. These figures represent the certainty (indirectly, the uncertainty factor) element involved with the estimates made. It is to note that, the estimates made would be certain if and only if the testing data is adequate enough to construct a Software Reliability Growth Model (SRGM). However, in case, where test history is not available or limited, estimates are made using subjective guess method. For these estimates to be more effective, one can associate a confidence/certainty factor with each estimate. Now, we try to look for an optimal solution for the same problem illustrated in the previous paragraph with the certainty factors in picture. Table III represents this case, here the numbers in column 3 (P_i) denotes the confidence values. Though, tc_6 has the highest improvement value, as the probability figure associated with it is comparatively much lesser than the confidence figure of tc_5 and also its reliability figure is not much lesser than tc_6 we should favour tc_5 over tc_6. Evidently, with the increase in the number of components, neither pen and paper methods nor heuristics based methods can address this scenario. Also, to the best of our knowledge none of the existing works are fit to handle the software reliability optimisation problem with uncertainty elements in play. The literature review presented in the Section VI strengthens our statement. Now, we proceed ahead with a formal definition of the problem and the solution methodology adopted.

TABLE III. **AN EXAMPLE WITH CONFIDENCE VALUES.**

Test suite	Ri_i	P_i	t_i
tc_1	0.05	0.7	7
tc_2	0.1	0.63	5
tc_3	0.15	0.5	5.5
tc_4	0.125	0.95	6
tc_5	0.195	0.9	8
tc_6	0.23	0.75	8

- Problem Formulation
  Given:
  - A set of software components, swc_1, swc_2 ...... swc_n.
  - A set of test suites, one test suite for each component, tc_1, tc_2,.... tc_n along with their execution times t_1, t_2,.... t_n.
  - Reliability improvement estimates for each test suite, Ri_1, Ri_2,.... Ri_n.
  - Probability/Confidence figure for each estimate, P_1, P_2, ... P_n.
  - The desired reliability of the system as R_d, and current reliability as R_{curr}.

Assumptions:

- Test suite is either tested fully or not executed at all (0-1).
- Time t_n indicates the total time taken to simulate/execute the test suite tc_n and
- Testing will improve the reliability of components.

Output:
Selection of a optimal set of test suites such that the reliability of the system meets the target/desired reliability R_d, having a minimum test execution time.

III. METHODOLOGY OVERVIEW

This section talks in detail about the solution methodology proposed to find an optimal set of test components to be targeted in attaining the desired reliability.

The problem ahead of us can be treated as a class of optimal testing-resource allocation problem concerned with allocating resources among several alternative (competing) options. The options are to be favored keeping the objective function in mind. Here, the objective function involves the cost factor together with reliability, which means that we have multiple objectives in terms of both maximizing system reliability and minimizing testing cost. If there is no uncertainty element involved in the problem domain, we can consider this problem as a multi-objective optimization problem whose solution is trivial: we would always select the test-suites having the maximum outcome, i.e., more reliability improvement in less time. The important point to note here is that, the environment before us now is not a deterministsic environment rather it is a probabilistic environment. It is a well known fact from the history of statistics that, the probabilistic environments are efficiently handled by learning and exploration is a necessary prerequisite of probability learning. The uncertainty about parameters drives learning and it is by exploration and by interactive interaction one can learn the behaviour of the system. Here, we summarise that some learning mechanism is indeed required to address the scenario.

Looking forward for the kind of learning theories that can be considered for, we see that learning under uncertainty can be well handled by *Reinforcement Learning* [3]. RL is a goal directed learning which gathers knowledge about the environment through interaction. Every interaction produces a wealth of information about the consequences of actions, and about what to do in order to achieve goals. The information gain over a number of interactions thus can be used to weed out the uncertainty element involved in and one can come up with a definite average consequence of chosing a particular option.

This is to say until we explore the selections a number of times, it is not trivial to know the true outcomes of each

selection. True outcomes of selecting an option can be made by observing the outcome in each trail and refining the expected outcome. It is worthy to note that exploration gives us a opportunity to learn more about the system, but, it may not result in high current rewards. Also, exploring all the time is not a good idea as it cannot give us a quick solution. In order to have a quick solution, it is always better to exploit the current knowledge on the estimates made. Exploitation is the right thing to do to maximize the expected reward on one step, but exploration may produce greater total reward in the long run. Whether we select an action either by exploration or by exploitation, the estimates of the selection are to be refined for every run so that they guide us our future action section policy. In any specific case, whether it is better to explore or exploit depends in a complex way on the precise values of the estimates, uncertainties, and the number of remaining steps. Such problems are paradigms of a fundamental conflict between making decisions that yield high current rewards, versus making decisions that sacrifice current gains with the prospect of better future rewards. However, exploration and exploitation approach will definitely result in a optimal solution.

We now discuss about how exploration and exploitation can be taken up after we introduce some related terminology.

1) *Action list, A* - The set of available choices represent the action list. Here, set of test suites tc_1, tc_2,.... tc_n represent the set of actions available.
2) *Reward, R* - The outcome of selecting a particular action. As our aim is to have more reliability gain in minimum time, we define the reward of selecting an action a as

$$R(a) = \frac{ReliabilityImprovement(Ri_a)}{TestExecutiontime(t_a)}$$

3) *Estimated Reward, $Q_n(a)$* - The estimate of the reward of an action a after n trials.
4) *True Reward, $q^*(a)$* - The true value of selecting an action a.

On a broader view, there are two basic steps needed:

- *Estimation* (the outcome of each action): Whether we explore or exploit, after the selection of every action, we need to refine the outcome of the action. In a simplistic way, action values are estimated/refined using sample-average method. Here, each estimate is an average of the sample of relevant rewards. By the law of large numbers, as the number of iterations increases, estimates converges to real values, i.e., *Qt(a)* converges to *q*(a)*.
- *Action selection policy*: Once the estimates are made, now comes the question of policy to be adopted to choose an action. One natural solution is to select the action having the highest estimated value, expressed as

$$A_t := argmaxQt(a)$$

If we maintain estimates of the action values, then at any time step there is at least one action whose estimated value is greatest. We call these the greedy actions. When we select one of these greedy actions, we say that we are exploiting our current knowledge of the values of the actions. If instead, we select one of the non-greedy actions, then we say we are exploring, because this enables us to improve our estimate of the non-greedy action's value. Greedy action selection always exploits current knowledge to maximize immediate reward; it spends no time at all in sampling apparently inferior actions to see if they might really be better. A simple alternative is to behave greedily most of the time, but every once in a while, say with small probability epsilon (ϵ), instead select randomly from among all the actions with equal probability, independently of the action-value estimates. These methods are coined as ϵ-greedy methods. An advantage of these methods is that, in the limit as the number of steps increases, every action will be sampled an infinite number of times, thus ensuring that all the *Qt(a)* converge to *q*(a)*.

A. Implementation and Performance Aspects

As discussed earlier, rewards are estimated using sample average method. Let R_i denote the reward received after the ith selection of an action a, and let Q_n denote the estimate of its action value after it has been selected *n - 1* times, which we can write simply as

$$Q_n := \frac{R_1 + R_2 + ... + R_{n-1}}{n-1} \quad (1)$$

The above equation can be devised as an incremental formula so that the averages can be updated with minimum computational costs. In other terms, the above equation can be expressed as

$$Q_{n+1} := Q_n + \frac{[R_n - Q_n]}{n} \quad (2)$$

Pseudocode for a complete algorithm using incrementally computed sample averages and ϵ-greedy action selection policy is shown in Algorithm 1. In the Algorithm 1, the function bandit(A) is assumed to take an action and return a corresponding reward.

Algorithm 1 epsilonGreedy

1: **Initialise:**
2: **for** a =1 to k **do**
3: $Q(a) \leftarrow 0$
4: $N(a) \leftarrow 10$
5: **end for**
6: **while** (1) **do**
7: $A = argmaxQ(a)$ *with probability 1-* ϵ
or
8: $A =$ *a random action, with probability* ϵ
9: $R \leftarrow bandit(A)$
10: $N(A) \leftarrow N(A) + 1$
11: $Q(A) = Q(A) + \frac{1}{N(A)}(R - Q(A))$
12: **end while**

The algorithm presented has been inspired by the multi-armed bandit solution strategies which is a simplistic form of reinforcement learning. To the best of our knowledge, we say this is the first effort to apply machine learning approach for software reliability optimisation.

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-668-2

IV. DEMONSTRATION USING REAL CASE STUDY

In this section, we present a real case scenario which forms the basis for our motivation to pursue this investigation. As an illustrative example, a relatively simple redundant system is considered, the configuration of which is depicted in Figure 1. The system is a heterogeneous dual redundant system comprising of a main system and a standby system. Each system houses two packages and each package in turn holds 3 sensors. Both Main and Standby systems acquires real time data from these sensors. The dynamics of main system and standby system are heterogeneous in nature. The main system is highly accurate, with less acceptable operational time, whereas the standby system is less accurate with longer operational time.

For the missions, whose working duration is of the order of the performance of main system, it is naturally good to consider the data of the main system so long it is healthy. As the data provided by this system is crucial from mission perspective, the redundancy management is employed at the component (sensor) level. If multiple component failure occurs then system level reconfiguration is considered. Component level redundancy management adds more to the complexity of the software logics coded. Thus, the functional correctness of redundancy management software logics play a significant role in determining the overall reliability of the system. The software architecture of such logics should be failure resilent and robust enough. From the verification perspective, it is of paramount interest to ensure the reliability of these logics. The section below details the software architecture considered for realising the component level redundancy.

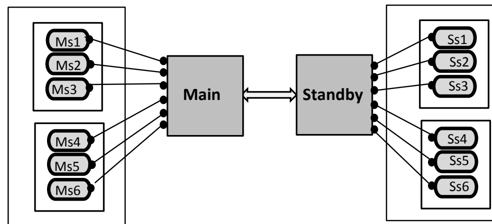

Figure 1. *System Configuration.*

A. Software Architecture

Redundancy management software core has 3 basic modules viz., i) Fault detection module ii) Fault Diagnosis module iii) Reconfiguration module. Fault detection logic recognizes that something unexpected has occurred in the system. To do so, it monitors the behavioral parameters of a component/system to asses the health of it. Once a fault is detected, the next step is to identify the faulty component (failure location), and also the nature of the fault (whether the fault is transient nature or permanent). Finally, a remedial action to be performed based on the decision logics of the system. This phase leads to reconfiguration of the system either at the component level or system level. Successful reconfiguration requires robust and flexible software architecture and the associated reconfiguration schemes.

Software realization for the system is depicted in Figure 2. In total, there are 12 modules labelled as swc_i. Each module is associated with a test suite (tc_i) and its execution time (t_i).

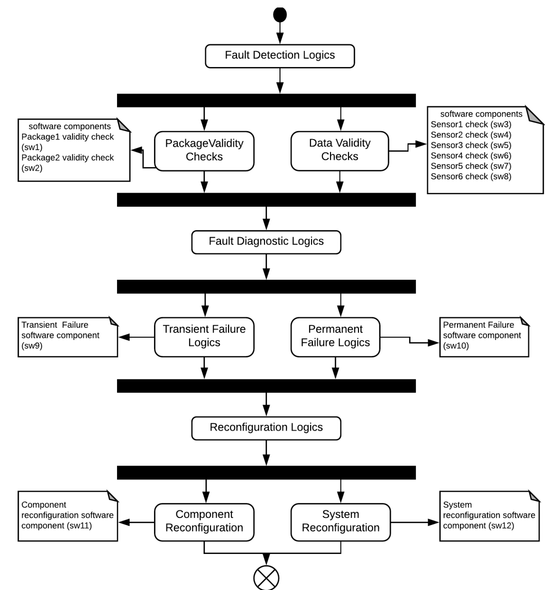

Figure 2. *Software Architecture.*

As stated earlier, each software component has its own contribution to the overall software reliability. The reliability improvement figure *Ri* upon testing different components is different. For the illustrated system, the *Ri* for each component is assigned based on the complexity and the functional criticality of the logic associated with it. In the Table IV, the 2nd row shows the *Ri* that can be obtained by testing a software component swc_i with its test suite tc_i. We see there is a variation in the *Ri's*. For example, software logics identified by swc_2 corresponds to package validity checks of the second package. For the system which is described above, the design elements for package2 are more in number when compared to package1, hence the logics are complex, as a result, the *Ri* is more for swc_2 when compared to swc_1. Similarly, swc_{10} deals with identifying permanent faults. As permanent faults lead to reconfiguration of the system, the decisions are made after observation of the fault for a persistent amount of time. Hence, this logic is obviously the most complex of all the components, hence the *Ri* awarded to it is the highest. 3rd row of the table refers the test suite execution times t_i. As our aim is to find an optimal test suite with minimum execution time, the reward *R* for each test component is

$$R_n = \frac{Ri_n}{t_n}$$

which are placed in 4_{th} row. 5th row corresponds to the uncertainties associated with this reward estimates. The uncertainties are assigned using subjective guess method. When there are more elements of uncertainty in the logics, the probabilities are assigned less. For example, the software logics for package validity checks, which are meant to asses the health of the sensor, depend on operational environmental conditions. The physical quantities like temperature of the chamber, number of operational hours, misalignment factors, calibration state of the package play a role in the sensor behaviour. As per the system design, package2 is highly sensitive and more dependent on the external factors. Hence, it is very difficult to ascertain *Ri* with great confidence, so the probability associated to it is on the lower side. On the other hand, there is no

Copyright (c) IARIA, 2018.

TABLE IV. **SOFTWARE COMPONENT NUMERICALS.**

Parameter	tc_1	tc_2	tc_3	tc_4	tc_5	tc_6	tc_7	tc_8	tc_9	tc_{10}	tc_{11}	tc_{12}
Reliability Improvement (Ri_i)	9	11.5	6	6	6	8	8	8	7.5	13	10	7
Execution Time *(ti)*	7	8	5	5	5	5	5	5	6	10	9	7
Reward *(R)*	1.285	1.4375	1.2	1.2	1.2	1.6	1.6	1.6	1.25	1.3	1.11	1.0
Probability (P_i)	0.75	0.65	0.65	0.65	0.6	0.6	0.6	0.6	0.9	0.85	0.9	0.9

involvement of external factors on the reconfiguration logics of the system. Only the design criteria as per system needs are to be coded, hence, the uncertainty factor associated to it is less. All components are assigned values following the same analogy.

V. EXPERIMENTAL RESULTS

This section demonstrates the evaluation results of the proposed algorithm to the case study illustrated in the previous section. In essence, two aspects are considered during evaluation. Firstly, the applicability of the bandit algorithm in arriving at an optimal solution is studied. To demonstrate this, 4 case studies each depicting an indicative practical scenario is taken up. Secondly, the aspect of exploration and exploitation tradeoff for the given task at hand is studied. Here, various exploration fractions are considered to arrive at a suitable exploration probability for the illustrated example. Also, on an explorative note, a simple variant of ϵ greedy algorithm termed as ϵ decaying strategy is applied. Here, rather than using a fixed value for ϵ, it is started with a high value initially, and decreased gradually over time. This way, we can favor exploration initially, and then favor exploitation later on. The following subsections elucidates the experimental results on a detailed note.

A. Application of Bandit Algorithm

To illustrate the application of bandit algorithm for finding an optimal test suite, four different scenarios/cases considering the different possibilities of having rewards and probability structures are taken. Each scenario results are illustrated using two graphs. The first graph depicts the preferred action choices and the second graph elucidates the reward history. The details of which are stated below:

- *Case1:* This case corresponds to the state where the rewards and probability figures of the case study are kept unchanged. The rewards and confidence figures depicted in Table IV are considered as it is. By intuition we can make some guess on the optimal test suite selection, but it may not be the possible best solution. By running the algorithm designed, we see the order of preference of test suite is as tc_9, tc_{10}, tc_{11}, tc_1, tc_6, tc_{12}, tc_7, tc_2, tc_8, tc_3/tc_4/tc_5. The *Ri's* of tc_6, tc_7, tc_8 are the highest(1.6), but the probabilities associated with them are less. Though the *Ri* of tc_9 is lesser than tc_6, since the associated probability of it is much more than tc_6, tc_9 is given preference. Also, we see the reward of tc_{10} is higher than the reward of tc_9 , but in long run tc_9 is given preference as the certainity factor of it is higher than tc_{10}. Similarly, all other test suites are preferred. The results shown in Figure 3 after running the epsilon greedy algorithm depict the optimal test suite selection for this case.
- *Case2:* This considers the scenario where the estimates made are certain, means there is no element of uncertainty. Thus, all the estimates are assigned a probability of 1 (equal probability distribution). In this case, the test suite having the highest initial estimate should be given preference. For our case, tc_6, tc_7, tc_8 are to be favored first. The graphs illustrated in Figure 4 confirm the expected notion.
- *Case3:* Here, we consider the case where the rewards obtained by selecting each action is equal, but each reward is having its own uncertainty factor. Naturally, the one having the highest probability of occurence should be given preference over the other. For illustration purpose, the initial rewards of all the test suites is set to a value of 1. Here, the test suites tc_9, tc_{11}, tc_{12} with the highest probability are to be favored. The graphs depicted in Figure 5 stand to this opinion.
- *Case4:* This case pertains to the scenario where there is no uncertainty in the estimates made and all components have equal reward. Ideally, in this case all actions are to be chosen with equal importance. The graphs presented in Figure 6 confirm the analogy.

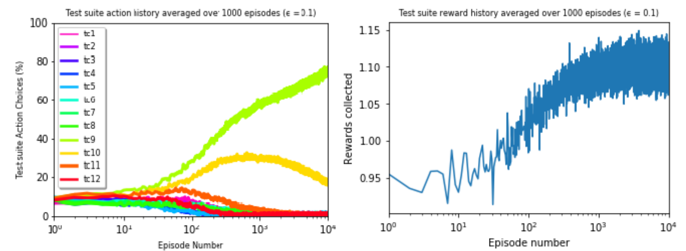

Figure 3. *Case 1 Results.*

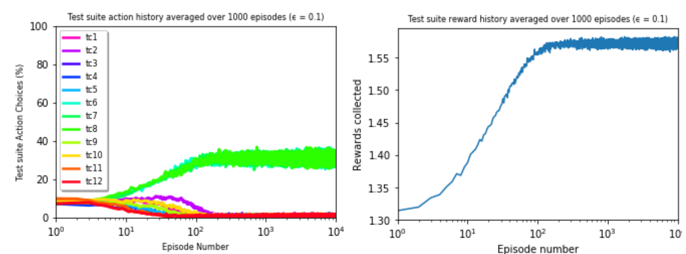

Figure 4. *Case 2 Results.*

The true reward of selecting a particular test suite besides the uncertainty factor in place can be concluded by considering the long term rewards. Table V summarizes the long term rewards R_l of every test suite obtained after running the proposed algorithm for the 4 different scenarios explained in the previous paragraph. For every case, row 1 corresponds to confidence figures, row 2 represents initial reward estimates and row 3 represents the true rewards of each test suite. We

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-668-2

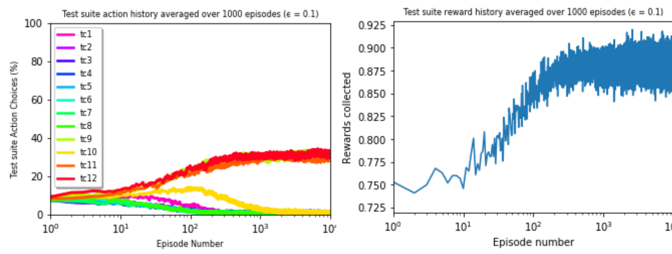

Figure 5. *Case 3 Results.*

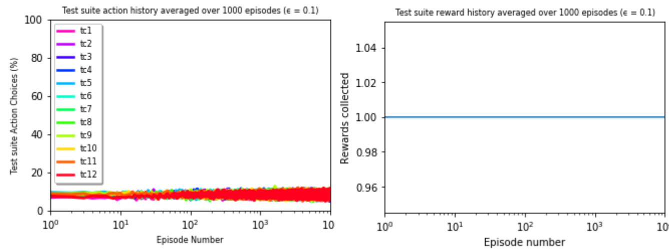

Figure 6. *Case 4 Results.*

see the long term rewards R_l's for each test suite entirely depends on the initial estimates on the probability of occurence of that estimate. The results depicted in Table V are found to be intuitive. Once the true rewards are known for each test suite, choosing an optimal set of test suites to fill the reliability gap is trivial.

B. Exploration-Exploitation Tradeoff

Now we look into the aspect of Exploration and Exploitation trade off by running the algoritm over various exploitation factors. Figure 7 compares a greedy method with three ϵ-greedy methods (ϵ = 0.01, ϵ = 0.1, ϵ = 0.5). All the methods formed their action-value estimates using the sample average technique. The greedy method improved slightly faster than the other methods at the very beginning, but then leveled of at a lower level. It achieved a reward-per-step of only about 1.21, compared with the best possible of about 1.26 on this testbed. The greedy method performed significantly worse in the long run because it often got stuck performing sub-optimal actions.The ϵ-greedy methods eventually performed better because they continued to explore and to improve their chances of recognizing the optimal action. The ϵ = 0.1 method explored more, and usually found the optimal action earlier, but it never selected that action more than 91% of the time. The ϵ = 0.01 method improved more slowly, but eventually would perform better than the ϵ = 0.1 method. As seen from the figures, exploring more often (ϵ = 0.5) is also not good. We say that this exploration-exploitation trade-off varies from problem to problem. For the given case study exploration rate of 0.01 is relatively effective in long run, but this needs more number of iteration to achieve sub-optimal rewards. Hence a choice can be made between exploration rate of 0.1 or 0.01.

One issue with the epsilon-greedy strategy is how to set the value of ϵ, and how we continue exploring suboptimal choices at that rate even after the algorithm identifies the optimal choice. Rather than using a fixed value for ϵ , we can start with a high value that decreases over time. This way, we can favor exploration initially, and then favor exploitation later on. This strategy is known as ϵ decaying strategy. Figure 8 and Figure 9 illustrates this strategy. The initial ϵ is set to 0.1. Figure 8 illustrates the case where the exploration factor is decreased to 0.01 after half the trials are over. We see that there is no performance degradation upon decreasing exploration rate as by half of the trials all the actions rewards might have reached very close to their optimal values, hence little exploration is enough. Figure 9 decreases the exploration factor by a large factor (to 0.001) which is not an advisable scenario for our case study. The initial choice of exploration rate and the decaying factor varies from problem to problem.

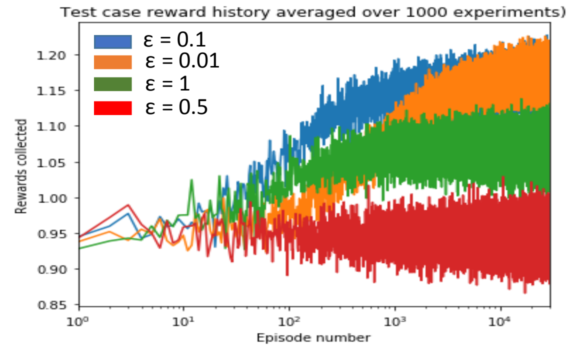

Figure 7. *Exploration-Exploitation tradeoff.*

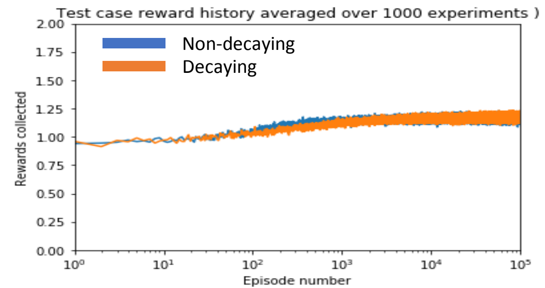

Figure 8. *Epsilon Decreasing Strategy-1.*

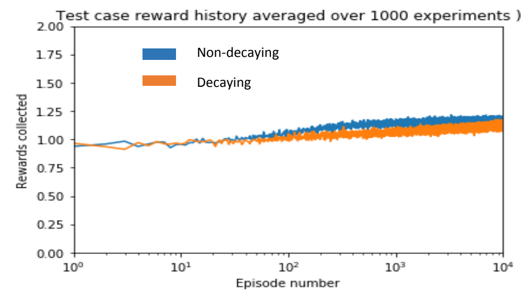

Figure 9. *Epsilon Decreasing Strategy-2.*

VI. RELATED WORK

A lot of work in the past considered the optimal allocation of the reliabilities to minimize a cost function, related to the design or the verification phase costs. Much initial research dealt with hardware systems (e.g., the series-parallel redundancy-allocation problem has been widely studied [4]-[5]); software systems received attention more recently. For a software application, the objective of the optimization will depend on the phase of the software life cycle. During the design phase [6], structural optimisation of the software architecture is paid attention with two leading objectives i) reliability constrained

TABLE V. **OVERALL RESULTS.**

Case No.	Parameter	tc_1	tc_2	tc_3	tc_4	tc_5	tc_6	tc_7	tc_8	tc_9	tc_{10}	tc_{11}	tc_{12}
1	P_i	0.75	0.65	0.65	0.65	0.6	0.6	0.6	0.6	0.9	0.85	0.9	0.9
	Ri_i	1.285	1.437	1.2	1.2	1.2	1.6	1.6	1.6	1.25	1.3	1.11	1.0
	Rl_i	0.96	0.93	0.78	0.78	0.78	0.95	0.95	0.95	1.11	1.09	0.99	0.90
2	P_i	1	1	1	1	1	1	1	1	1	1	1	1
	Ri_i	1.285	1.437	1.2	1.2	1.2	1.6	1.6	1.6	1.25	1.3	1.11	1.0
	Rl_i	1.28	1.43	1.2	1.2	1.2	1.59	1.59	1.59	1.25	1.3	1.11	1.0
3	P_i	0.75	0.65	0.65	0.65	0.6	0.6	0.6	0.6	0.9	0.85	0.9	0.9
	Ri_i	1	1	1	1	1	1	1	1	1	1	1	1
	Rl_i	0.75	0.65	0.65	0.65	0.65	0.6	0.58	0.58	0.88	0.84	0.88	0.88
4	P_i	1	1	1	1	1	1	1	1	1	1	1	1
	Ri_i	1	1	1	1	1	1	1	1	1	1	1	1
	Rl_i	1.0	1.0	1.0	1.0	1.0	0.99	1	1	1	1	0.99	1

cost minimization ii) cost constrained reliability maximization. During the testing phase, the objective of the optimization is to determine the allocation of testing effort so that the desired reliability objective is achieved [7][8]. During the operational phase, optimization is used to explore alternative configurations and to determine an optimal allocation of components to various nodes in a distributed network to achieve the desired performance and reliability.

At the software front, we see much of the research work in the community is biased on design phase optimisation side [9][10]. This work, however, does not consider testing-time of software components and the growth of their reliability. Not many papers considered the problem in the software verification phase, where the issue is either to allocate reliabilities that components need to achieve during their testing or to determine the allocation of testing effort so that the desired reliability objective is reached. Looking at the work in this direction, chronologically, the very initial work by Okumoto and Goel [11] investigated the optimal software release problem by using a software reliability growth model based on NonHomogeneous Poisson Process (NHPP) by considering the cost and software reliability as two different independent criterion. This work was carry forwarded by Shigeru Yamada et al. [12] who consider both cost and reliability criteria.

Also, authors in [7][13] proposed an optimization model with the cost function based on well known reliability growth models. They also include the use of a coverage factor for each component, to take into account the possibility that a failure in a component could be tolerated (but fault tolerance mechanisms are not explicitly taken into account, and the coverage factor is assumed to be known). The authors in [14] also try to allocate optimal testing times to the components in a software system (here, the reliability growth model is limited to the Hypergeometric (S-shaped) Model). Some of the cited papers [7][14][15] also consider the solution for multiple applications, i.e., they aim to satisfy reliability requirements for a set of applications.

In recent times, traditional reliability growth modelling techniques are replaced by machine learning techniques to improve the prediction accuracy of the constructed SRGM. A number of machine learning strategies such as artificial neural networks (ANN), support vector machine (SVM) and genetic programming (GP), are in practice in recent times for reliability modeling. Gene Expression Programming (GEP), a new evolutionary algorithm based on Genetic algorithm (GA) and GP, has been acknowledged as a powerful ML for reliability modelling[16]. We infer from the literature survey that the application of AI in software engineering domain [17] is also concentrated on software relaibility modelling.

Though, SRGM is probably one of the most successful techniques in the literature for software reliability modelling, with more than 100 models existing in one form or another, through hundreds of publications, in practice, however, they encounter major challenges. First of all, software testers seldom follow the operational profile to test the software, so what is observed during software testing may not be directly extensible for operational use. Secondly, when the number of failures collected in a project is limited, it is hard to make statistically meaningful reliability predictions. Thirdly, some of the assumptions of SRGM are not realistic, e.g., the assumptions that the faults are independent of each other, that each fault has the same chance to be detected in one class, and that correction of a fault never introduces new faults. These limitations impediments the use of SRGM based methods. In such cases, subjective knowledge of the system can be taken as an aid in making the reliability estimates. Estimates can be made subject to some criteria like - complexity, functional criticality etc. Since these are just estimates, one can assign confidence factor for the estimates made and can address the optimal selection issue.To the best of our knowledge, no work addresses this scenario and hence motivated us to consider a learning strategy and frame a solution methodology.

VII. CONCLUSIONS

In this paper, we have proposed a learning based paradigm for addressing the software reliability optimisation problem. We have shown how an interactive learning can address the problem of finding an optimal test suite whose rewards are stochastic in nature. In the proposed solution, every interaction is used to learn about the system and in a way the rewards are refined. As a result, over a period of time, the stochastic reward values are converted into true rewards. Once the true rewards are computed, the problem at hand becomes as simple as a multi-objectuive optimisation problem. The learning strategy employed here is inspired by a well known multi-armed bandit solution strategies. The application of the proposed solution strategy to the real case study demonstrates the potential of Reinforcement Learning in addressing the problem stated.

In future, we intend to enhance our existing algorithm by considering various practical scenarios. The random action selection policy considered during exploration phase in ϵ-greedy solution can be improved using Upper-Confidence-Bound strategy. The ϵ greedy action selection forces the

non-greedy actions to be tried, but indiscriminately, with no preference for those that are nearly greedy or particularly uncertain. It would be better to select among the non-greedy actions according to their potential for actually being optimal, taking into account both how close their estimates are to being maximal and the uncertainties in those estimates. We find there is a proper mathematical basis to work out this idea. Using this refined startegy, we guess that global optimum can be obtained in a fewer iterations. Furthermore, based on some numerical preference Gradient Bandit Algorithms can be considered to improve further. Furthermore, contextual bandits can be explored for associative search policies. In summary, we see a good potential in the application of RL for optimisation domain.

REFERENCES

[1] M. R. Lyu, “Software reliability engineering: A roadmap,” in 2007 Future of Software Engineering, ser. FOSE ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 153–170. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.24

[2] S. S. Gokhale, “Architecture-based software reliability analysis: Overview and limitations,” IEEE Transactions on Dependable and Secure Computing, vol. 4, no. 1, Jan 2007, pp. 32–40.

[3] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. The MIT Press, 2017.

[4] Y. Nakagawa and S. Miyazaki, “Surrogate constraints algorithm for reliability optimization problems with two constraints,” IEEE Transactions on Reliability, vol. R-30, no. 2, June 1981, pp. 175–180.

[5] F. A. Tillman, C. L. Hwang, and W. Kuo, “Determining component reliability and redundancy for optimum system reliability,” IEEE Transactions on Reliability, vol. R-26, no. 3, Aug 1977, pp. 162–165.

[6] M. E. Helander, M. Zhao, and N. Ohlsson, “Planning models for software reliability and cost,” IEEE Transactions on Software Engineering, vol. 24, no. 6, Jun 1998, pp. 420–434.

[7] M. R. Lyu, S. Member, S. Rangarajan, and A. P. A. V. Moorsel, “Optimal allocation of test resources for software reliability growth modeling in software development,” IEEE Transactions on Reliability, 2001.

[8] J. Rajgopal and M. Mazumdar, “Modular operational test plans for inferences on software reliability based on a markov model,” IEEE Transactions on Software Engineering, vol. 28, no. 4, Apr 2002, pp. 358–363.

[9] F. Zahedi and N. Ashrafi, “Software reliability allocation based on structure, utility, price, and cost,” IEEE Trans. Softw. Eng., vol. 17, no. 4, Apr. 1991, pp. 345–356. [Online]. Available: http://dx.doi.org/10.1109/32.90434

[10] O. Berman and N. Ashrafi, “Optimization models for reliability of modular software systems,” IEEE Transactions on Software Engineering, vol. 19, no. 11, Nov 1993, pp. 1119–1123.

[11] K. Okumoto and A. L. Goel, “Optimum release time for software systems,” in Computer Software and Applications Conference, 1979. Proceedings. COMPSAC 79. The IEEE Computer Society’s Third International, 1979, pp. 500–503.

[12] S. Yamada and S. Osaki, “Cost-reliability optimal release policies for software systems,” IEEE Transactions on Reliability, vol. R-34, no. 5, Dec 1985, pp. 422–424.

[13] M. R. Lyu, S. Rangarajan, and A. P. A. van Moorsel, “Optimization of reliability allocation and testing schedule for software systems,” in Proceedings The Eighth International Symposium on Software Reliability Engineering, Nov 1997, pp. 336–347.

[14] R.-H. Hou, S.-Y. Kuo, and Y.-P. Chang, “Efficient allocation of testing resources for software module testing based on the hyper-geometric distribution software reliability growth model,” in Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on, Oct 1996, pp. 289–298.

[15] N. Wattanapongsakorn and S. P. Levitan, “Reliability optimization models for embedded systems with multiple applications,” IEEE Transactions on Reliability, vol. 53, no. 3, Sept 2004, pp. 406–416.

[16] B. Kotaiah and R. A. Khan, “A survey on software reliability assessment by using different machine learning techniques,” 2012.

[17] M. Harman, “The role of artificial intelligence in software engineering,” in 2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE), June 2012, pp. 1–6.