Expert Systems With Applications 198 (2022) 116722

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

# Intelligent software debugging: A reinforcement learning approach for detecting the shortest crashing scenarios

Engin Durmaz *, M. Borahan Tümer[1]

*Marmara University, Faculty of Engineering, Computer Engineering Department, Egitim, Fahrettin Kerim Gokay St., 34722, Kadikoy, Istanbul, Turkey*

ARTICLE INFO

*Keywords:*
Reinforcement learning in automated bug detection
Exploring crashes by SARSA
Exploring crashes by prioritized sweeping
Delta debugging
Detected goal catalyst

ABSTRACT

The Quality Assurance (QA) team verifies software for months before its release decisions. Nevertheless, some crucial bugs remain undetected in manual testing. These bugs would make the system unusable on field, thus merchant loses money then manufacturer loses its customers. Thus, automatic software testing methods have become inevitable to catch more bugs. To locate and repair bugs with an emphasis on the crash scenarios, we present in this work a reinforcement learning (RL) approach for finding and simplifying the input sequence(s) leading to a system crash or blocking, which represents the goal state of the RL problem. We aim at obtaining the shortest input sequence for the same bug so that developers would analyze agent's actions causing crashes or freeze. We first simplify the given crash scenario using Recursive Delta Debugging (RDD), then we apply RL algorithms to explore a possibly shorter crashing sequence. We approach the exploration of crash scenarios as a RL problem where the agent first attains the goal state of crash/blocking by executing inputs, then shortens the input sequence with the help of the rewarding mechanism. We apply both model-free on-policy and model-based planning-capable RL agents to our problem. Furthermore, we present a novel RL approach, involving Detected Goal Catalyst (DGC), which reduces the time complexity by avoiding grappling with convergence via stopping learning at a small variance and attaining the shortest crash sequence with an algorithm that recursively removes the unrelated actions. Experiments show DGC significantly improves the learning performance of both SARSA and Prioritized Sweeping algorithms on obtaining the shortest path.

## 1. Introduction

Automated bug detection methods are inevitable because both the number of software applications and the code size grow faster than the size of quality assurance (QA) teams. In other words, it becomes impossible to manually create test cases to cover all software for each version. User Acceptance Testing (UAT) phase consists of check integration testing, check test strategy document, check integration sign-off, repair, and coordinate release (Lawanna, 2012). This cycle would repeat many times due to changes in the repair step. During this process, an in-depth analysis of each part of code added to the new software release by the QA team is not feasible. Even if engineers in development and quality teams can follow each updated procedure in software, they would miss a bug that results from a combination of various flows. As a result, automated bug detection methods are required for both helping the debugging process of developers and defining new test cases for QA teams.

In this part, we emphasize the importance of automated black box testing for software released. Automated tests are applied to software in different ways, such as static code analysis, unit testing, integration testing and black box testing. Pattern based static methods analyze paths of codes to determine bugs. However, for bug detection, writing patterns for massive bodies of code does not make sense because it is not easy to cover the entire program (Pradel & Sen, 2017). In addition, maintenance of patterns is difficult because the original program is subject to change every day. Unit testing, one of the most widely used automated testing methods, is used to detect bugs when any part of software is changed. Meanwhile, unit tests would not be enough to locate the cause of failing flows since developers have limited the number of scenarios to verify the source code (Singh & Singh, 2019). Although all unit tests are run successfully, it is possible that the combination of functions would not work properly. Therefore, integration tests are applied as another approach in automated software testing. On an integrating process, the number of dummy components written should be as small as possible to minimize the cost (Traon et al., 2000). On each software release, this difficult process should be performed carefully so that the corresponding errors would not occur

* Corresponding author.
*E-mail addresses:* engin.durmaz@live.com (E. Durmaz), borahan.tumer@marmara.edu.tr (M.B. Tümer).
[1] The advisor

https://doi.org/10.1016/j.eswa.2022.116722
Received 19 July 2021; Received in revised form 15 February 2022; Accepted 21 February 2022
Available online 7 March 2022

in the field. Even though all test cases are passed during unit and integration tests, a crash scenario is not captured if the bug scenario consists of multiple cases lacking a relationship directly between them. Therefore, in addition to unit and integration tests, software should also be verified with system testing in which a QA team evaluates how the various components in the System Under Test (SUT) interact together and interact with other parts of the system. Test engineers usually do not know what codes and libraries the SUT consists of. Instead, they verify the system in several ways, such as by typing inputs, by clicking buttons via interfaces, and so on. The method proposed in this study provides this feature of black-box testing to generate crash scenarios.

This study proposes black-box testing methods based on reinforcement learning (RL) where the RL agent learns from interaction with software being tested how to find the shortest sequence of keys/clicks that causes a system crash/freeze. In the context of RL, the RL environment is software, the actions are the keys pressed, the policy is the sequence of keys/clicks and the goal state is a system crash/freeze. After many runs, the agent should learn the best policy, which is the sequence of actions that maximize the total expected reward. The test cases of crashes are obtained in this RL based automated testing. When a RL agent generates a crash or an exception scenario, developers become capable of locating a solution for the bug by repeating the test case. Since the algorithms are implemented from the perspective of black-box testing, the RL agent remains independent of the details of the code to identify the code that caused the bug. This approach meets the minimum complexity condition and the verification process becomes easily maintainable. So even if the code has experienced a major modification between two versions, RL algorithms do not require any change.

Before RL training, we apply Recursive Delta Debugging to simplify a given long sequence. The Delta Debugging algorithm isolates the relevant variables and values by systematically narrowing the state difference between a passing run and a failing run—by assessing the outcome of altered executions to determine whether a change in the program state makes a difference in the test outcome (Zeller, 2002). By using the logging mechanism of the SUT, we obtain the sequence that consists of inputs from the startup of the program to crash time. When the obtained sequence is too long, it takes too much time to localize bug location. Thus, we use a recursive version of delta debugging to eliminate irrelevant parts of this sequence. When the simplified sequence is still not sufficiently short enough to analyze the crash reason, we should explore new crashing sequences shorter than this sequence. In this work we show that RL algorithms can find new sequences out.

The goal of this study is not only to find a crash sequence but also to shorten the sequence found because developers may likely reject dealing with longer sequences that are impractical to reproduce the bug (Mao et al., 2016). For this reason, we developed *Crash Detection Module* (CDM) in which both the model-free and model-based planning-capable RL algorithms, SARSA and prioritized sweeping (PS) in respective order, are implemented to explore the shortest crashing sequence. Lots of related work is found in literature regarding the detection of bug scenarios (Koroglu et al., 2018; Riganelli et al., 2020). One of the main differences from them is that we, instead of solving a crash problem of a specific system, share a more general form. The algorithms in this study are applicable to a variety of software, including mobile, desktop and service applications. In addition, we propose the Detected Goal Catalyst (DGC) solution, a novel approach to RL based testing. In this form, RL algorithms run to decrease the length of crashing sequence until the variance of last episodes reaches a threshold value of $threshold_{dg}$, then our algorithm progressively removes unrelated actions to find an exact scenario. Experiments show that this mechanism shortens the time required for obtaining the shortest path of a crash.

The novelties and features that distinguish our work from the state-of-the-art can be summarized as in the following. First, this study concentrates on obtaining the shortest input sequence(s) resulting in crashes and blocking cases by sending generated test inputs to entry points of application under test. Second, the implementation of our approach can be adapted not only to graphical user interfaces but also to embedded applications. Finally in this work, we present the Detected Goal Catalyst (DGC) algorithm, a novel extension to RL on simplification of crash scenarios, that allows the RL agent to stop training in DGC threshold, $threshold_{dg}$, which is higher than the RL threshold. DGC achieves the exact crash sequence in a more straightforward fashion by a variant of local search which does not take as long as the remaining RL episodes. Since RL algorithms spend most of their time around the goal state to converge on the exact length of the crash sequence, we make significant improvements in execution time (reductions up to 91% for SARSA and 24% for PS) by leaving RL training earlier, but close to episode convergence value.

The rest of this paper is organized as follows. Relevant work in the literature is reviewed in Section 2 where differences between this study and them are also listed. We continue with Section 3 where some background work is shared. Then, we lay out the methodology in Section 4 where we discuss how both SARSA and PS are employed in automated software testing followed by a subsection about how DGC functions and is integrated to both RL algorithms. Section 5 consists of experiments and evaluation criteria. Finally, we conclude the study in Section 6.

## 2. Related work

Automated bug detection is not a new topic. Since predefined test cases are insufficient to cover software bugs, methods are required to generate bug scenarios automatically. Mobolic, one example of this approach, combines the online testing technique and customized input generation, in this way, tests with high coverage are automatically generated (Arnatovich et al., 2018). EvoSuite (Fraser & Arcuri, 2011), based on an evolutionary search approach, uses Java byte code of the SUT to generate test suites that achieve high code coverage. Another study, EvoMaster, uses an evolutionary algorithm to automatically generate test cases for RestfulAPI (Arcuri, 2019). Researchers in Panichella et al. (2018) developed DynaMOSA which used a many-objective GA tailored to the test case generation problem. The studies mentioned concentrate on code coverage and they almost present solutions for only Object-Oriented applications. However, in this study, our target is not only Object Oriented applications. Instead, we developed a black-box testing module that uses the input set of an SUT to find and simplify crashing scenarios.

Crash reproduction utilities help developers debug bug-related codes and enlarge test suites with new test cases. RECORE uses evolutionary search-based test generation to reconstruct failures from saved core dump (Röß ler et al., 2013). RECORE uses the similarity measurement idea of EvoSuite shared above. Another stack trace based study is STAR (Chen & Kim, 2015) that statically analyzes the crashed methods using backward symbolic execution, which is both path and context sensitive. MuCrash, another approach (Xuan et al., 2015), takes the stack trace in the crash, source code, and existing test cases as input, then applies mutations to produce new test cases. In other work, JUnit test cases necessary to run on the SUT to cause the bug were automatically produced using stack traces belonging to a existing crash (Nayrolles et al., 2017). EvoCrash (Soltani et al., 2020b), an extension of EvoSuite, used Guided Genetic Algorithm (GGA), which leverages the stack trace to guide the search toward generating tests able to trigger the target crashes. A benchmark based evaluation (Soltani et al., 2020a) showed EvoCrash was successful in reproducing the majority of crashes (more than 75%) from Commons-lang, Commons-math, and Joda-Time. Botsing, an open-source, extendable search-based framework (Derakhshanfar et al., 2020), to reproduce the crash takes as input a stack trace, and SUT. As it is seen, these studies use stack traces of the exception to reconstruct the same failure. And they almost concentrated only on the applications based on Object-Oriented architecture. In this

study, we do not use the stack trace of an existing exception. Our proposed module first explores a new crashing input sequence for crash and then removes unrelated inputs to obtain a simpler sequence.

Black-box testing methods (Wang et al., 2013) based on automated test case selection approach are to find uncertainties. They compare test duration, and execution failures with older ones to help test case prioritization (Spieker et al., 2017). Then, quality assurance verifies first these most important cases. CrashScope (Moran et al., 2017), Sapienz (Mao et al., 2016), Mobicomonkey (Ami et al., 2018) are other tools that use program inputs to generate a bug report, which contains number of crashes and test sequence lengths. Alternatively, we propose an approach which concentrates on obtaining the shortest input sequence for a crash or a freeze. The *Crash Detection Module (CDM)* we developed in this study makes selections from the input set to find a bug scenario. Then, *CDM* explores a shorter sequence that causes the same bug.

A large branch in automated test case generation for software verification is fuzz testing, which uses malformed/semi-malformed data injection in an automated fashion to find bugs. A group of fuzzing tools such as Hwacha (Choi et al., 2018), ASTAA (Hutchison et al., 2018), and RobOT (Wang et al., 2021) focuses on automatic robustness testing. To increase code coverage, FuzzGen (Ispoglou et al., 2020) performs a whole system analysis, iterating over all programs and libraries that use the target library. Since tracing code coverage is costly, UnTracer (Nagy & Hicks, 2019) proposes an idea of coverage-guided tracing targeted at reducing the overheads of coverage-guided fuzzers. In this study, we approach the bug detection problem from a different perspective than these Fuzzing techniques. For instance, our proposed module does not behave as an attacker to catch vulnerabilities of the system. Our RL based module tests the system as an end-user to discover crash scenarios that make the system unusable. When a test case is found, our module does not stop. Instead, CDM tries eliminating unrelated parts to obtain simplest sequence, which still causes the same bug.

Recently, RL methods are applied to online testing in the test case selection phase. WebExplor, a study on online testing, adopts curiosity-driven RL to generate high-quality action sequences (test cases) satisfying temporal logical relations (Zheng et al., 2021). In another game testing mechanism, a cloud-based malware detection scheme in the dynamic game (Xiao et al., 2017), mobile devices apply the Dyna architecture to emulate its planning and reactions from hypothetical experience. The main difference in our study is that our method is not a type of online testing. Since the bugs causing an application to crash prevent end-users from using the application, online testing is not a good solution for crashes in products in the field. Therefore, in this study, we reveal the crash scenarios in the testing phase of the software development life cycle.

Another group of studies to generate test cases using an RL approach focuses on locating functions which are related to bugs. Wei et al. (2015) developed EvoQ, a study based on evolutionary RL, which uses reflection to extract objects and methods to create an initial population. Then each individual is evaluated using a fitness function based on the execution trace. The QTIP, another test case generation technique, converts the library version of SUT to an RL environment then invokes functions of the SUT (Kim et al., 2018). In contrast to these studies, our algorithm does not use an application function set or application states. Instead, *Crash Detection Module* uses the input set of the SUT as actions of the RL agent.

RL algorithms are used in automated-black box testing. QLearning-Based Exploration (QBE) which is a fully automated black-box testing methodology explores GUI actions using QLearning (Koroglu et al., 2018). They obtain models of Android applications from a training set by executing them with Random Exploration strategy and they create Q-Matrix. In addition, they use the Q-Matrix to generate test suites. Another model-based RL testing study builds a behavioral model gradually and generates test cases based on the model (Vuong & Takada, 2018). Another study on Android GUI testing is a test generation algorithm based on Q-learning to systematically select events and explore the GUI of an application under test without requiring a preexisting abstract model (Adamo et al., 2018). In this study, we do not concentrate on GUI testing. The *CDM* uses a valid and known input set to obtain simple crashing sequences. Furthermore, our target is not limited only to Android applications but also any other type of applications is possible.

The most influential study on input simplification which can be applied to arbitrary input without having any a priori knowledge about the test case format is delta debugging algorithm (Zeller, 2002; Orso et al., 2006). An improved version of delta debugging algorithm, Recursive Delta Debugging (RDD), is more robust and provides faster and more accurate results (Kiss et al., 2018; Fevotte & Lathuilière, 2019). RDD efficiently solves the crash simplification problem. However, in case the simplest crashing sequence does not exist in the given crashing sequence, the RDD algorithm is insufficient. RDD does not generate new crashing sequences that include a possible shorter crashing sequence than the simplified version of the given sequence. Thus, in this study, we apply RL algorithms to explore shorter scenarios of the same bug.

In this section, we have shared the differences between our work and previous studies. The novelties of this research are listed in the Introduction section.

## 3. Preliminaries

This section briefly describes the main concepts that form the background for this study. We first list the types of errors leading to crashes. Then automated black-box testing is shortly explained. Then, we share a brief discussion on Temporal Difference (TD) learning and SARSA, and Prioritized Sweeping (PS) that represent various approaches in RL. Finally, we explain the RDD algorithm.

### 3.1. Crash reasons

The main purpose of the automated testing approach presented in this paper is to explore input sequences that result in crash or freeze. For this reason, we first define error types that cause application locking or restart.

Commercial software data from 30 organizations such as Aspect Security, Breach Security, CERT, Homeland Security, Microsoft, MITRE, Oracle, Red Hat shared with SANS institute to identify the most common and dangerous 25 software failures (Rawat & Dubey, 2012). Each of these dangerous problems as illustrated in Fig. 1 changes the program behaviors at run time. This paper concentrates on exploring scenarios of control flow errors, which are listed as follows, in contrast to finding the problem's exact location in code.

**Null usage** is a one type of reason for crash. Null usages account for 37.2%–41.7% in the memory bugs (Li et al., 2006). These unassigned usages should be detected before delivery of software. **Buffer overflow** and **stack overflow** are the other important reasons for crashes. Buffer overflow occurs when the write operation overrides values in addresses adjacent to the destination buffer. CERT declared buffer overflows as a security risk (Ruwase & Lam, 2004). For these reasons, they should be determined before version update in production. Stack overflow implies that depth of the stack has reached an uncontrolled range. Stack overflows should be detected during the software verification. In addition to these, **unhandled errors** affect program flow adversely. Crash reports collected from Google Play show that uncaught exceptions occur in more than 70% of crashes (Lenarduzzi et al., 2018). Moreover, freeze reasons, for instance, **endless loops**, would cause the application to not respond to user actions anymore. The scenarios that put the application in an unresponsive state should be detected.

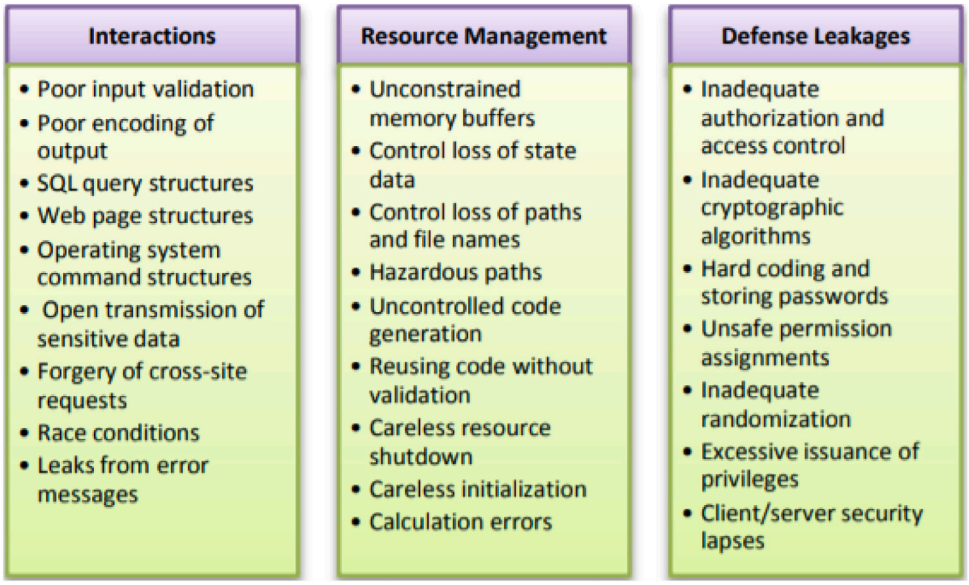



**Fig. 1.** In year 2008 and 2009, SANS institute classified most dangerous 25 problems into three groups (Rawat & Dubey, 2012).

### 3.2. Automated black-box testing

In this study, we apply black-box testing approach in automated testing. Using external testers may not be technically and economically feasible most particularly in embedded systems (Chen & Dey, 2001). Automated testing is also inevitable for high level applications because time and memory space as critical resources are usually insufficient to manually validate each test case of large systems. Here are some of the advantages we get by speeding up the testing process with automation:

- A shorter software development cycle
- Frequent releases
- Quicker changes and updates to the application
- Faster time-to-market delivery

In contrast to usual automated testing in which procedures are randomly selected from a collection, we do not define any procedure containing test cases in our mechanism. We assume the application as a black-box and we only know the input set of the application. In view of this mechanism, our test modules inject input instances generated in run time as actions in RL algorithms.

### 3.3. Temporal Difference (TD) learning and SARSA

TD learning is a model-free RL approach where action value adjustments contain an estimating component based upon the future (destination) states so per-step (or intra-step) adjustments are possible as opposed to Monte Carlo adjustments only at the ends of episodes (Sutton & Barto, 2018). TD learning is a combination of Monte Carlo methods and Dynamic Programming in the sense that, like MC methods, TD learning directly utilizes raw experience without a model of the environment's dynamics while, as opposed to them, making use of a per step estimate of the target value TD methods are able to update action values at each step in an episode.

TD methods require waiting only until the next time step. In $one-step$ TD method, state value $Q(S_t, a_t)$ is updated with discounted next state value $Q(S_{t+1}, a_{t+1})$ and with observed reward $R_{t+1}$ at time t+1. In SARSA, return values for previous states are calculated with learning rate, $\alpha$ and contribution of next state values are obtained by multiplying with a discount rate of $\gamma$. Both the learning rate and the discount rate are in the range of [0, 1]. The agent of $SARSA$ algorithm interacts with the environment and its value updates are based upon actions taken at destination states (i.e., $S_{t+1}$) as in Eq. (1). Hence this flow is known as an $on-policy$ learning algorithm.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{1}$$

### 3.4. Prioritized Sweeping (PS)

In most of the real-world cases, the state action space is tremendously large, thus, minimizing unnecessary value updates in the model increases the performance of the algorithm. PS is a model-based RL technique which can be used for Markov prediction (Moore & Atkeson, 1993). In PS, only the most "important" actions' values are updated by a priority metric that prioritizes each individual update regarding size of the adjustment of the attempts to measure the anticipated size of adjustment for each action value. In the PS algorithm, after taking an action if the reward is bigger than $\theta$ then it is put to queue. Actions in the queue are consumed until the queue is empty. If the queue is empty, actions are $\epsilon$-greedy selected.

$$p \leftarrow |r + \gamma max'_a Q(s', a') - Q(s, a)| \tag{2}$$

A PS agent employs a partial model of the environment to keep track of state–action pairs executed during the flow of episodes in the learning process. It updates multiple actions' values backtracking over these paths of state–action pairs where previous actions predicted to lead to the current state become potential candidates for getting values updated. The PS agent uses two mechanisms to avoid superfluous value updates: first it prioritizes the value updates among those candidate actions by the size of update as in (2) and it updates only those actions' values with an update size over a predefined threshold. Second it uses a parameter $N$ for the backtracking depth to update the values of a multiple number of actions so as to speed up learning. With $N = 1$ PS turns to on-policy or off-policy TD learning. Hence, this adjustment based model allows PS to outperform other model-free (e.g., SARSA and Q learning) and model-based (e.g., Dyna) approaches (Sutton & Barto, 2018).

### 3.5. Recursive Delta Debugging

Delta Debugging approach submitted in Zeller (2002) applies divide-and-conquer algorithm to simplify given input. The main goal is to isolate problem relevant parts when a program which does not work correctly is given. In Mozilla crash problem in Orso et al. (2006), delta

debugging minimized an original 896-line HTML input which caused the Mozilla browser to crash.

To obtain more accurate results in bug scenario simplification, we apply Recursive Delta Debugging (RDD) as in Algorithm 1. If we have a long sequence that causes a crash, this algorithm eliminates irrelevant parts of the crashing sequence. At the end, we obtain the simplest crashing path of the given sequence.

**Algorithm 1** Recursive Delta Debugging

**rDDMin** *(initial, pre, post, validator)*

**if** length of $initial == 1$ **then**
  **return** $initial$
**end if**
$left \leftarrow subset\ of\ initial\ from\ 0\ to\ midd$
$right \leftarrow subset\ of\ initial\ from\ midd\ to\ end$
**if** $validator.check(pre + left + post)$ **then**
  **return** $rDDMin(left, pre, post, validator)$
**end if**
**if** $validator.check(pre, right, post)$ **then**
  **return** $rDDMin(right, pre, post, validator)$
**end if**
$part1 \leftarrow rDDMin(left, pre, right + post, checker)$
$part2 \leftarrow rDDMin(right, pre + first, post, checker)$
**return** $part1 + part2$

**EndFunction**

## 4. Methodology

Here we present the *Crash Detection Module (CDM)*. A system diagram of CDM is given in Fig. 2. CDM produces actions and sends them to the SUT. When SUT does not return a response in time, CDM decides that a crash has occurred. CDM continues to learn the system to find the shortest input sequence.

CDM is designed to discover the input sequence stopping the application in the validation process. CDM first by applying the RDD algorithm simplifies a randomly generated crashing input sequence, then the RL agent in CDM looks for shorter sequences. CDM has one model-free and one model-based planning-capable module, SARSA and PS, respectively. CDM may be configured to run with any of these modules. In addition to these modules, the Detected Goal Catalyst (DGC) algorithm has been developed to find an exact crash scenario faster. DGC can be applied to both SARSA and PS modules. We finally discuss how CDM can be integrated with real world applications.

In this study, the application is assumed as a black box. We first designed a general state diagram for possible crash scenarios of applications as shown in Fig. 3. This diagram does not show the application state logic, instead it shows RL state transitions in the shortest crash sequence problem. The diagram shows the input set of $A$ executable by an application and the crash scenario $C$ consisting of different actions $c_j$ where $c_j \in A$. Each action $c_j$ moves application crash state from $s_j$ to $s_{j+1}$. For this reason, only $c_j$ is important for RL agent at state $s_j$. In this study, we want to detect the sequence composed of $c_j$s, $\forall c_j \in C$.

The proposed method in this study can be considered a kind of fuzzing, which is a highly automated software testing technique. Fuzzing is a robustness-testing method where generated inputs are sent to SUT. The purpose of fuzzing is to better ensure the absence of exploitable vulnerabilities by covering the boundary cases in the application's logic (Liljedahl, 2019). In this study, we do not analyze SUT to determine specific input sequences. For instance, our module does not try sending values which are the upper bound of a function or which are larger than an array defined in SUT. Our module behaves as an end-user of the system. This module performs single input in each action decision. At the end of each episode, we obtain a list of inputs whose composition creates a crash. In this way, this module catches crash scenarios, which belong to unrelated system functions and which are executed in different states of SUT.

### 4.1. Defining the shortest crash sequence problem in the RL framework

We designed a RL approach for crash scenario detection. Input of SUT, such as sequences of keypad keys, button clicks, and so on, form the state-independent action space of this RL problem. At each step of an episode, the agent selects an action from within this action space and executes it as in Algorithm 3. Each input key (i.e., an action), is concatenated to form an agent state. That is, each possible sequence formed by the concatenation of input keys (i.e., elements of the action space) is a state in the RL problem. The agent interacts with testing software to reach the goal state, i.e., a crash/blocking. If SUT does not turn back for the action executed, this implies that the agent reaches the crash/blocking state. Then, the last action reaching the goal state is rewarded as 1. The rewarding mechanism of RL makes the crash sequence shorten in the coming episodes. At the end of an episode, we calculate the variance of last episodes' lengths and we compare it with a prespecified parameter value to understand whether the convergence criterion is achieved or not as in Algorithm 2. Once the variance of the length of crash sequences the RL algorithm detects in the final $k$ episodes falls below a prespecified threshold value the learning process terminates. The shortest sequence found by episodes in CDM is the shortest crash scenario and it helps developers fix the bug more quickly.

**Algorithm 2** PS for the Shortest Crash Sequence Problem (uses runPSEpisode in Algorithm 3)

**Function FindCrashSequenceByPS**
**inputs:**
  $IS$ ▷ input set of SUT = actions agent can take
  $env$ ▷ environment to be sent actions/inputs
$stLearned \leftarrow rDDMin(initalSequence)$ ▷ simplify with RDD.
$bestCS \leftarrow NIL$ ▷ current best crashing sequence
$seqLengths$ ▷ list of length of found crash sequences
$set\ all\ indices\ of\ seqLengths\ to\ MAX$
$episodeCounter \leftarrow 0$
**while** $episodeCounter < maximum\ episode\ count$ **do**
  $env.prepare()$
  $lastCS \leftarrow$ **runPSEpisode**$(env, stLearned, IS)$ ▷ runPSEpisode algorithm is shown on next page
  **if** $length\ of\ lastCS < length\ of\ bestCS$ **then**
    $bestCS \leftarrow lastCS$
  **end if**
  $put\ length\ of\ lastCS\ to\ seqLengths$
  $\sigma \leftarrow calculateVariance(seqLengths)$
  **if** $\sigma < convergence\ threshold$ **then**
    **break** ▷ learning done
  **end if**
  $episodeCounter \leftarrow episodeCounter + 1$
**end while**
**return** $bestCS$

We implemented both SARSA and Prioritized Sweeping (PS) algorithms in CDM. The implementation of the PS method of the CDM module is shared in Algorithm 3. SARSA algorithm in the CDM is similar to PS except prioritized queue implementation.

### 4.2. Detected goal catalyst

As a termination condition the downfall of the variance below the prespecified threshold value is critical for the finalization of learning. Experiments show that variance decreases sharply at first, but it slows down in later phases toward the end of learning. The variance of the estimated crash sequence length of the latest episodes changes slowly as shown in the sample run in Fig. 4-a. For this reason, we developed the Detected Goal Catalyst (DGC) algorithm, which first stops the RL method at a higher threshold value (i.e., for a premature level of convergence) as in Fig. 4-b. Then, irrelevant actions are removed from the best sequence found by the RL agent. This approach finds the exact

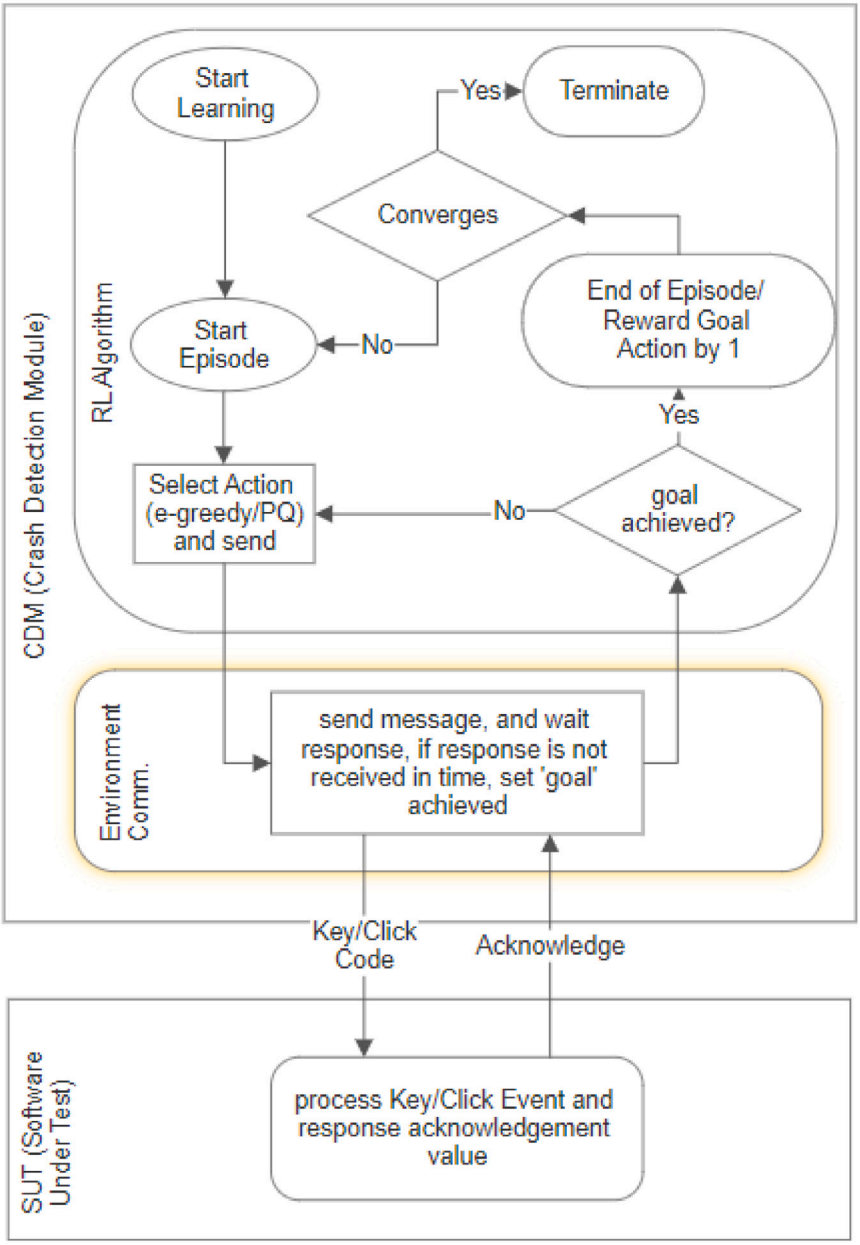



**Fig. 2.** Block Diagram of Crash Detection Module.

crash sequence faster. In this study, DGC is adapted to both SARSA and PS methods.

The main idea in the DGC approach is estimating whether the RL agent is sufficiently close to the goal state (i.e., the exact length of crash scenario). To estimate the distance from the exact crash scenario the RL agent employs the variance of sequence lengths at the last $k$ episodes. When the variance is smaller than a dynamic threshold value, $threshold_{dg}$, the agent stops RL iterations as in algorithm 4. Then, the agent executes the SearchCrash algorithm 5 on the best sequence obtained in RL episodes to attain the exact sequence. In the first step of the algorithm 5, the agent removes the first action from the best sequence and it executes the actions in the remaining part by sending them to the environment. Then, the agent checks whether the remaining part still causes a crash. If the remaining part also causes a crash, the agent removes the action from the best sequence. If the remaining part does not result in a crash, the best sequence is not updated. Then, the second iteration starts where the agent selects the next action to test. By testing each action in the resulting sequence at the end of the RL algorithm to understand whether it has a role in crash or not, the algorithm 5 gradually reveals all actions of the best sequence and removes the superfluous actions of no influence to crash.

**Mechanism of Detected Goal Catalyst:** In this solution, the regular execution of the selected RL algorithm is switched to DGC as the variance of the crash sequence length falls down below $threshold_{dg}$.

An example scenario is shared to explain better. In this scenario, the action set of the program is $A = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T\}$ and the crash sequence is $C = [C\text{-}R\text{-}A\text{-}S\text{-}H]$. The RL agent has reached goal state with [M - **C** - A - **R** - P - **A** - **S** - *T* - **H**] actions when threshold of $threshold_{dg}$ is achieved. The length of the sequence is 9 and the proposed algorithm removes the irrelevant actions as in Table 1.

The $threshold_{dg}$ value should not be too high to escape RL episodes terminating early. Because, the time complexity of calculations to obtain exact crash sequence depends on the length of the best scenario found in RL episodes. In addition, this threshold value cannot be constant since the length of exact crash sequence is unknown. For this

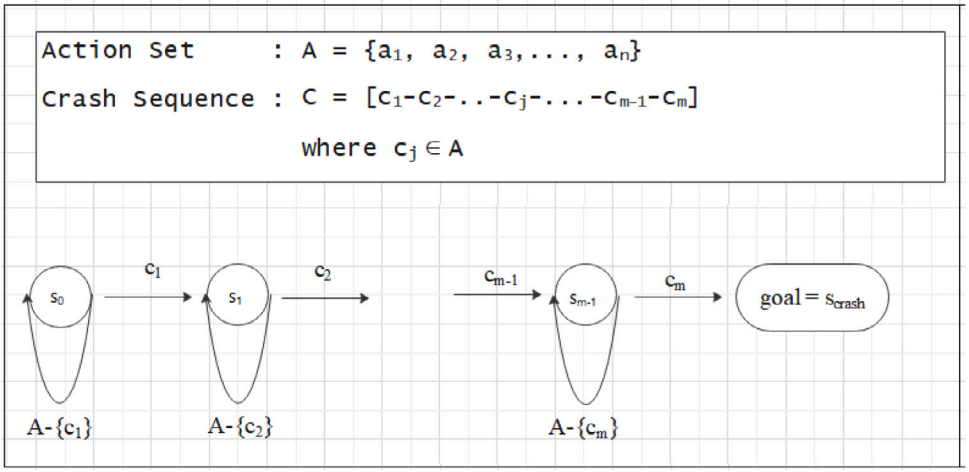
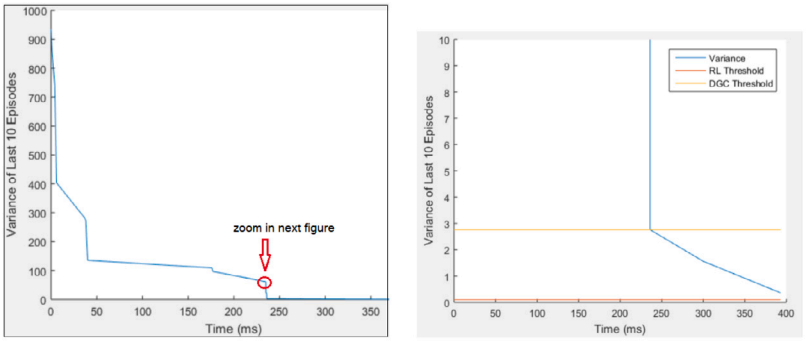



**Fig. 3.** State Diagram of Crash Scenarios.




(a) Overall episodes in Sarsa (b) DGC threshold bigger than RL threshold

**Fig. 4.** Defining DGC threshold to save time.

**Table 1**
Iterations of Crash Sequence Extraction for "C-R-A-S-H" in "M-C-A-R-P-A-S-T-H".

| Nr | Description |
|---|---|
| 1 | Initialize environment. The first routine of recursive DGC starts with M - C - A - R - P - A - S - T - H and Ignore the first pivot action of "M". Remaining sequence is sent to the environment. And it results in a crash. Thus new best sequence is **C** - A - **R** - P - **A** - **S** - T - **H** |
| 2 | Initialize environment. Remaining sequence starts with "C". When "C" is removed, the crash does not occur. Thus "C" is kept. |
| 3 | Initialize environment. Next action is "A", when A is removed and actions are executed the crash is reproduced again. Thus new sequence is **C** - **R** - P - **A** - **S** - T - **H** |
| 4 | Initialize environment. When A is removed, "R" is selected as the pivot. If "R" is removed, the crash is not achieved. So, "R" is saved. |
| 5 | Initialize environment. Pivot now points to "P", when it is removed, the agent can repeat crash. Then "P" is removed. Remaining sequence is **C** - **R** - **A** - **S** - T - **H** |
| 6 | Initialize environment. The action "A" is a new pivot. When "A" is removed, the crash condition is not satisfied. Thus, "A" is kept. |
| 7 | Initialize environment. The next action is "S". When the remaining sequence is sent to the environment, it will not cause a crash. Then, the agent keeps the "S". |
| 8 | Initialize environment. Now, "T" is pivot action. It is removed. And, the remaining part causes a crash. Thus new sequence is **C** - **R** - **A** - **S** - **H** |
| 9 | The end of the initial sequence actions is reached. And, best sequence is decided as **C** - **R** - **A** - **S** - **H** |

reason, the agent can decide this value by referencing the number of episodes past.

$$threshold_{dg} \leftarrow log_b(episodeCounter/100)x0.1 + b \quad (3)$$

The threshold value starts with a small number, then it increases logarithmic as in Eq. (3). In this formula, $b$ is the base of the logarithm function, and $episodeCounter$ is the number of episodes the agent passed. The value of $b$ depends on the System Under Test (SUT) and it can be empirically found. The *CDM* runs without DGC configuration, and learning is completed for a crash scenario. At the end, $b$ value is calculated as the degree of root. For example, assume the test terminates after running 10000 episodes, and the exact crash length is found as 5. In this case, $log_b(10000/100)x0.1 = 5$, then $b$ is equal to 1.09. The $b$ value can be used for other scenarios of the SUT. The equation contains $+b$ term, because learning should be finished if the agent finds the shortest path.

### 4.3. Integration of Crash Detection Module with real applications

*Crash Detection Module* (CDM) may be injected in the main loop of existing programs when *CDM* is linked to the application as a library. In the case of unsuitable program code for this approach, *CDM* runs outside and sends commands via debug port. *CDM* is adaptable to both embedded and high-level applications.

*CDM* may also adapt to various types of applications such as console applications, GUI (graphical User Interface) applications and service applications. Because the requirements for the integration are only the

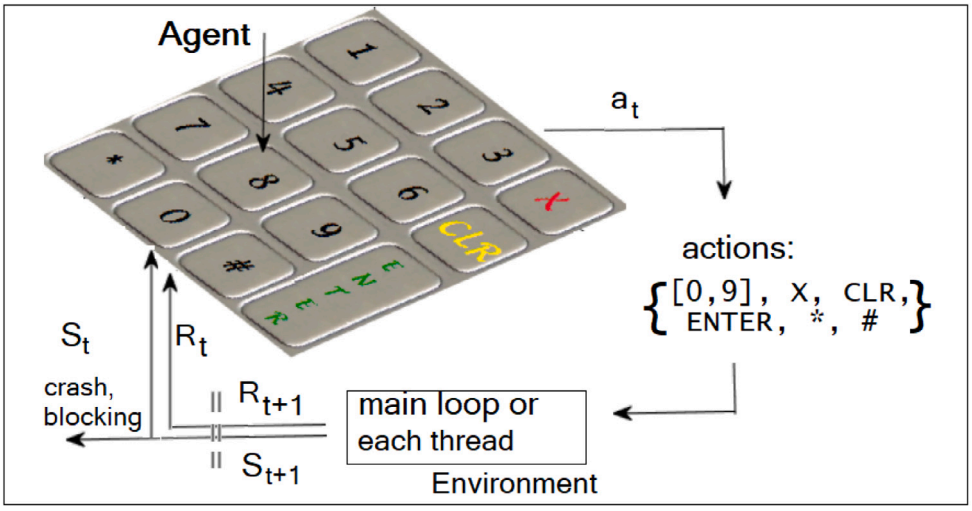



**Fig. 5.** Crash Detection Module : Agent generates keys to stop application.

**Algorithm 3** Episode of Prioritized Sweeping for the Shortest Crash Sequence Problem

**Function runPSEpisode**
**inputs:**
$IS$ ▷ input set of SUT = actions agent can take
$env$ ▷ environment to be sent inputs of SUT
$stLearned$ ▷ list for observed states
**output:**
$CS$ ▷ crash sequence found in the episode

$CS := empty$ ▷ no input in crash sequence at the start
$s_t := initialize\ state$ ▷ $s_t$ : current state
**if** $stLearned\ is\ not\ empty$ **then**
  $s_t \leftarrow first\ state\ of\ stLearned$
**end if**
$a_t \leftarrow selectActionGreedy(s_t, IS)$ ▷ $a_t$ : input for SUT
**while true do** ▷ loop until goal state (crash) occurs
  $isGoal \leftarrow env.sendToSUT(a_t)$
  $add\ a_t\ to\ crash\ sequence\ list\ of\ CS$
  **while true do** ▷ loop for prioritized queue
    $r_t := isGoal?1 : 0$ ▷ only goal is rewarded
    **if** $isGoal$ = **true then** ▷ goal is reached
      $\delta := r_t + \gamma * 0 - Q(s_t, a_t)$ ▷ no next state
      $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha * \delta$
      **break** ▷ end of episode
    **end if**
    $s_{t+1} \leftarrow loadState(s_t, a_t)$ ▷ $s_{t+1}$ : next state
    $\delta := 0$
    $maxP_a \leftarrow s_{t+1}.getPriorAction()$
    **if** $maxP_a == -1$ **then** ▷ no prior set before
      $maxP_a \leftarrow getMax(s_{t+1})$ ▷ Find Max(s', a')
      $maxPQ \leftarrow s_{t+1}.getActionReward(maxP_a)$
      $p \leftarrow r_t + \gamma * maxPQ - Q(s_t, a_t)$
      **if** $p > \theta$ **then**
        $s_{t+1}.setPriorAction(maxP_a)$
      **end if**
    **end if**
    **if** $maxP_a == -1$ **then** ▷ still no prior action
      **break** ▷ exit p queue loop
    **end if**
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha * \delta$
    $s_t \leftarrow s_{t+1}, a_t \leftarrow maxP_a$
    $isGoal \leftarrow env.sendToSUT(a_t)$ ▷ Prior Queue loop checks isGoal first
    $add\ a_t\ to\ crash\ sequence\ list\ of\ CS$
  **end while**
  **if** $isGoal$ = **true then** ▷ goal is reached
    **break** ▷ end of episode
  **end if**
  ▷ continue learning, select next action of next state
  $a_{t+1} =\leftarrow selectActionGreedy(s_{t+1}, IS)$
  $\delta \leftarrow r_t + Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$
  $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha * \delta$
  $s_t \leftarrow s_{t+1}, a_t \leftarrow a_{t+1}$
**end while**
**return** $CS$
**EndFunction**

input set accepted by program and communication protocol such as Transmission Control Protocol (TCP) or Serial port. We see in Fig. 5 how the input sequences of a Point of Sale (POS) application become actions of reinforcement learning clearly. RL agent produces corresponding values of these keyboard keys and it sends the values to the application. And the application consumes the key. We do not change the application. We only add a TCP or Serial (RS232 or USB) listener to the application to get the keys which are sent from the *CDM*. Algorithm 6 is a general main loop of console and embedded applications. User key clicks are consumed in the main loop of application. Integration of *CDM* to SUT can be implemented as it is seen in Algorithm 7. Then, *CDM* can send the keys produced by the RL agent to SUT. When the program under test returns nothing, the *CDM* decides the program has corrupted. This means the agent reached the goal state. Thus, the last action is rewarded with 1. Then the next episode starts. And following episodes, the crashing input sequence is shortened.

If the device does not have such interfaces, the *CDM* can be used as a library. In this case, the application under test registers a key listener to the *CDM* and the *CDM* triggers this listener to send actions generated by the agent. When the software in the device crashes, the device could be restarted automatically by boot loader. When the device is restarted, the *CDM* understands the goal state has been reached. Thus, rewards of actions are calculated and a new episode is started.

**Algorithm 4** Integration of Detected Goal Catalyst to Prioritized Sweeping

**Function findCrashSequenceByPSWithDGC**
**inputs:**
*IS* ▷ input set of SUT = actions agent can take
*env* ▷ environment to be sent actions/inputs
*stLearned* ← *rDDMin*(*initalSequence*) ▷ simplify with RDD.
*bestCS* ← *NIL* ▷ current best crashing sequence
*seqLengths* ▷ list of length of found crash sequences
*set all indices of seqLengths to MAX*
*episodeCounter* ← 0
**while** *episodeCounter* < *maximum episode count* **do**
  *env.prepare*()
  *lastCS* ← **runPSEpisode**(*env*, *stLearned*, *IS*) ▷ runPSEpisode algorithm is shown on previous page
  **if** *length of lastCS* < *length of bestCS* **then**
    *bestCS* ← *lastCS*
  **end if**
  *put length of lastCS to seqLengths*
  *σ* ← *calculateVariance*(*seqLengths*)
  **if** *σ* < *convergence threshold* **then**
    **break** ▷ learning done
  **end if**
  *tDGC* ← 1.5 ▷ tDGC: DGC threshold which determines if RL training is enough and jump to local search, it is updated by episode counter
  *dgcLogFactor* ← *log*(*episodeCounter*/100, 1.5)
  *tDGC* ← *dgcLogFactor*/10 + *tDGC*
  **if** *σ* < *tDGC* **then**
    *bestCS* ← *searchCrash*(0, *bestCS*, *env*)
    **break** ▷ best sequence calculated
  **end if**
  *episodeCounter* ← *episodeCounter* + 1
**end while**
**return** *bestCS*
**EndFunction**

### 4.4. Performance evaluation of algorithms

CDM consists of three main flows on simplifying a crash sequence: RDD, DGC, and RL. In this section, we compute the complexities of these algorithms.

The time complexity of SARSA is found as $O(E * I * A)$, where E denotes the number of episodes, I the number of iterations in each episode, and A the number of actions for $\epsilon - greedy$ selections. On complexity analysis of PS, we consider actions selected in a prioritized queue. In this study, the length of the queue *CDM* employed in PS tests is 1. Then, time complexity of PS is found $O(E * I' * A) + O(E * P)$. Here, $I'$ denotes the number of iterations where $\epsilon - greedy$ selection is applied and $P$ denotes the number of iterations where actions are selected in a prioritized queue. The time complexity of the RDD, which uses Divide-and-Conquer algorithm is $O(N * log(N))$ where $N$ is the length of the first crashing sequence. The complexity of the DGC is $O(D)$ where $D$ is the size of the crash sequence obtained at the end of RL episodes. Since $D$ is not bigger than $N$, $O(D)$ is not a determinant of complexity. Finally, the total time complexities of crash sequence simplification with SARSA and PS are shared in Eq. (4) and in Eq. (5), respectively.

$$O(CDM_{SARSA}) = O(max(N * log(N), E * I * A)) \quad (4)$$

$$O(CDM_{PS}) = O(max(N * log(N), E * I' * A + E * P)) \quad (5)$$

The space complexity of the SARSA is $O(A * S)$ where A is the number of actions and S is the number of states. In PS, we use a variable for prioritized action of each state. Thus, space complexity of PS is $O((A + 1) * S)$. In addition, *CDM* to calculate variance of the last 100 episodes allocates memory with the size of $V$. RDD, and

**Algorithm 5** Removing Actions Irrelevant to Crash

**Function searchCrash**(*start, bestCS, env*)
**inputs:**
*start* ▷ index in bestCS, pivot position
*bestCS* ▷ current best crashing sequence
*env* ▷ environment to be sent actions/inputs
**output:**
*newBest* ▷ best crashing sequence found by search
*newBest* := *array*[*bestCS.length* − 1] ▷ we are looking for a sequence exactly one input smaller in each recursive call
*shorterFound* ← **false**
**for** *pivot* = *start* to *bestCS.length* **do**
  *env.prepare*()
  **for** *j* = 0 to *newBest.length* **do** ▷ actions except pivot are sent to environment
    *target* ← *j*
    **if** *j* ≥ *pivot* **then**
      *target* ← *target* + 1
    **end if**
    *newBest*[*j*] ← *bestCS*[*target*]
    *isGoal* ← *env.sendToSUT*(*newBest*[*j*])
    **if** *isGoal* = **true then**
      *shorterFound* ← **true**
      **break**
    **end if**
  **end for**
  **if** *shorterFound* = **true then**
    *start* ← *pivot* ▷ continue from this index
    **break**
  **end if**
**end for**
**if** *shorterFound* = **true then** ▷ continue with newBest
  *nextShorter* ← **searchCrash**(*start*, *newBest*, *env*)
  **if** *nextShorter*! = *NIL* **then**
    *newBest* ← *nextShorter*
  **end if**
**else**
  *newBest* ← *NIL*
**end if**
**return** *newBest*
**EndFunction**

**Algorithm 6** General Main Loop of Software

**Function Main** *(args)*
*pressedKey* ← 0
*initializeDrivers*()
*registerKeyHandler*(*onKeyPress*)
**while** *exit* ≠ 1 **do**
  **if** *pressedKey*! = 0 **then**
    *processKey*(*pressedKey*)
    *pressedKey* ← 0
  **end if**
**end while**
**return** 0
**EndFunction**
**Function onKeyPress** *(key)*
*pressedKey* ← *key*
**EndFunction**

**Algorithm 7** Integrating Crash Detection Module over Serial/TCP

**Function Main** *(args)*

```
pressedKey ← 0
initializeDrivers()
registerSerialHandler(onRLCmd)
registerTCPHandler(onRLCmd)
while exit ≠ 1 do
    if pressedKey! = 0 then
        processKey(pressedKey)
        sendResponseRL()
        pressedKey ← 0
    end if
end while
return 0
```

**EndFunction**

**Function onRLCmd** *(cmd)*

```
pressedKey ← extractKey(cmd)
```

**EndFunction**

DGC algorithms do not require extra space, because they work on the memory for the crashing sequence. As a result, $O(A * S) + O(V)$ and $O((A+1) * S)+O(V)$ are the total space consumption of SARSA and PS, respectively, during the shortest crashing input sequence exploration.

## 5. Experiments and discussion

Performance of both SARSA and PS on simplification of crash sequences have been analyzed by a series of experiments. Furthermore, another set of experiments have been conducted to study the effect of DGC on the time complexity of these algorithms. Results obtained on both sets of experiments are discussed in this section.

### 5.1. Evaluation metrics

The first metric in measuring the performance of any algorithm supported by the *Crash Detection Module* (CDM) is the estimation accuracy of the length of shortest sequence found by RL agent. This value should be equal to the length of the crash sequence. Each algorithm is run 100 times for each test scenario. Average elapsed time of the tests is reported after multiple runs.

Other metrics for learning performance assessment are the number of episodes and elapsed time on convergence. To decide upon convergence, the variance of the last 100 episodes' lengths is calculated. When variance is smaller than 0.1, the algorithm exits with success code. There are also limits for the maximum number of episodes to escape infinity. The maximum number of episodes for a test is equal to the multiplication of crash sequence length and 10000.

Algorithm parameters listed in Table 2 were decided after empirical runs of various combinations. For instance, $\theta$ parameter of PS was decided as 50 which best fits for the POS crash problem as shown in Fig. 6. If the DGC parameter is enabled, it is not activated until the variance of the last 100 episodes decreases to $threshold_{dg}$ value. The $threshold_{dg}$ value is calculated using Eq. (6). Increments in episode counter affect the threshold value logarithmically.

$$threshold_{dg} \leftarrow log_{1.5}(episodeCounter/100)x0.1 + 1.5 \quad (6)$$

### 5.2. Experiments

We first performed tests using the SARSA and PS algorithms to find crash sequences of different environments. Second, we enabled the DGC flag and repeated the tests. The performance improvements over both SARSA and PS modules were observed when DGC was enabled. Moreover, we implemented an RDD-based random crash sequence generator, RandRDD, and we compared the performance of PS and RandRDD. The PS algorithm showed better performance than RandRDD. The performance of PS as discussed later in depth was up to 44% better than that of RandRDD when DGC was enabled.

**Table 2**
Parameters in experiments.

| Parameter Name | Value |
|---|---|
| Reward to Goal State Action | 1.00 |
| Alpha (Learning Rate) | 0.20 |
| Gamma (Discount Factor) | 0.80 |
| Theta Single (PS Threshold) | 0.25[a] |
| Theta Multi (PS Threshold) | 0.50[a] |
| Epsilon (Balance exploration/exploitation) | 0.20 |
| Convergence Variance | 0.10 |

[a]If problem has one best solution, theta was set to 0.25. if problem has multiple best solution, theta was set to 0.50.

**Table 3**
Input sequence of the virtual environment.

| Alpha Chars. | Numeric Chars. |
|---|---|
| A-B-C-D-E-F-G-H-I-J-K-L-M | 1-2-3-4-5 |
| N-O-P-Q-R-S-T-U-V-W-X-Y-Z | 6-7-8-9-0 |

**Table 4**
Exact crash sequences for each experiment.

| TN | Crash sequences |
|---|---|
| 01 | "D-O-U-B-L-E-B-U-G-S" or "W-B-U-G-S" |
| 02 | "L-O-G-5-O-F-3-S-U-B-3" or "L-O-G-4-O-F-0" |
| 03 | "S-Q-R-T-9-9-S-U-B-1-0-0" or "S-Q-R-T-N-E-G-1" |
| 04 | "5-D-I-V-8-S-U-B-6-A-D-D-2" or "5-D-I-V-2-S-U-B-2" |
| 05 | "S-I-N-P-I-D-I-V-2-D-I-V-C-O-S-P-I-D-I-V-2" or "T-A-N-P-I-D-I-V-2" |

The RL modules and other parts of the application used in the experiments were implemented by Java (version of 1.8.0.231) programming language. The application also has a console option to run and to print results. Available actions and crash sequences from these actions are generated randomly. The application finds a crash sequence applying selected RL algorithm and DGC flag.

Each sequence was tested with four algorithms: SARSA without DGC, PS without DGC, SARSA with DGC, and PS with DGC. Each of these combinations was repeated 100 times and we calculated the average elapsed time for each of these 100 runs. All tests were first run on Dell (Latitude, 3500) computer. To check the hardware platform dependency, the same tests were also executed using a Lenovo (Ideapad, 720s) laptop. We observed that the relative performance of algorithms are identical on these machines. We only share the detailed test outputs obtained on Dell.

In experiments, we performed RL tests on two environments that have crash scenarios. The first environment model was a crashing sequence environment. The second environment was a simplified Point of Sale, POS, environment.

#### 5.2.1. Test with crashing sequence environment

This environment accepts alphanumeric characters listed in Table 3 as the program's valid inputs. Each input of the program is also an action of the RL agent.

We used 5 different virtual sequence environment models in tests. Each model has two crashing sequences and these are shown in the corresponding row of Table 4. An environment model throws a crash exception when all entries in one of its crashing sequences are executed.

Initial crashing sequences of these experiments are listed as follows:

- Initial crashing input sequence of the 1st experiment:
  Q-V-2-Z-O-K-Q-Y-T-4-4-7-D-U-O-2-P-4-S-7-E-K-U-3-M-N-O-2-M-5-3-N-7-A-7-8-J-X-2-Z-C-Q-B-P-9-6-7-D-1-2-6-K-

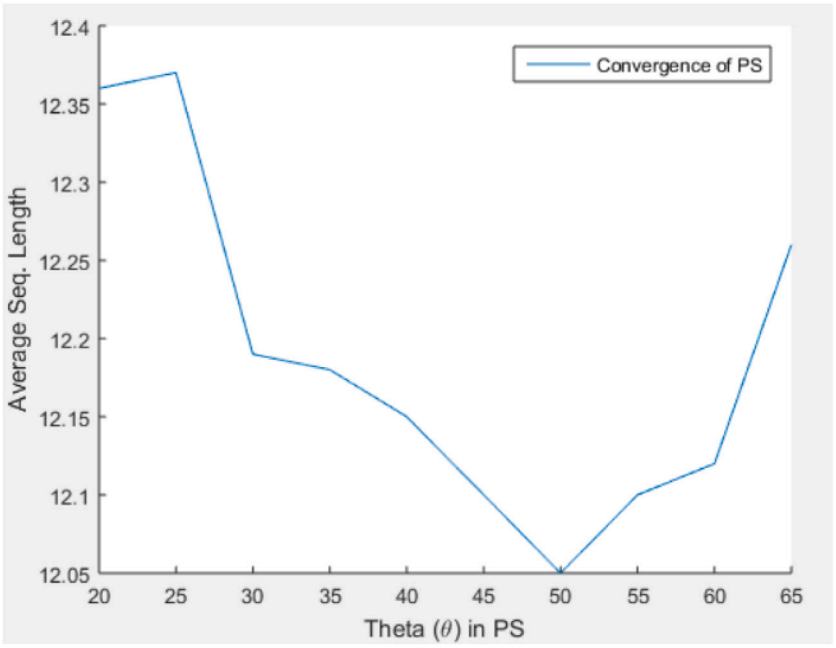



**Fig. 6.** $\theta$ value selection in POS crash test 5 where the optimum length was 12. The value of 50, which satisfied PS to converge the optimum crash sequence length was chosen.

B-Y-3-8-E-Z-Z-T-[L]-[E]-5-E-0-D-I-G-[B]-7-8-3-7-1-1-7-7-Q-C-0-4-0-K-H-D-1-T-F-V-L-K-S-L-I-Z-6-5-8-Z-P-D-6-P-R-0-Z-V-8-D-T-R-3-K-G-R-Q-2-M-5-M-Z-B-H-8-3-M-P-7-5-8-Q-Y-[U]-1-D-X-L-P-L-0-O-6-H-O-I-D-I-J-W-F-Y-N-Z-D-1-2-J-8-8-X-Y-N-W-7-L-2-1-A-C-H-B-W-3-Z-W-C-1-E-6-W-N-O-R-K-[G]-N-M-1-L-L-8-U-N-A-[S]

- Initial crashing input sequence of the 2nd experiment:
7-F-P-A-7-9-B-2-E-6-W-6-1-S-J-O-E-R-9-M-[L]-H-G-I-X-H-N-R-R-N-D-Q-8-A-G-V-[O]-R-3-M-K-P-K-Q-D-N-4-Q-R-I-S-V-W-S-5-2-V-6-4-6-5-1-4-2-7-U-H-T-E-3-M-7-3-R-3-9-F-T-4-N-0-0-B-U-0-6-7-4-A-Q-0-[G]-[5]-M-G-S-U-Z-X-[O]-E-8-6-V-O-M-P-M-0-9-X-R-I-1-K-B-0-M-0-N-U-V-P-L-Z-X-9-H-Y-E-2-H-6-W-Q-J-6-S-2-2-A-5-5-Z-0-6-Z-M-5-L-E-5-Z-5-D-9-6-3-G-5-U-5-L-S-8-W-G-J-N-K-Z-[F]-F-I-W-X-9-P-[3]-D-T-6-W-[S]-B-F-V-A-Y-0-S-C-K-6-Q-8-F-G-V-P-E-[U]-N-U-N-[B]-S-O-7-N-1-H-L-C-A-9-F-M-R-E-E-U-K-T-X-M-N-M-X-6-0-E-O-M-X-X-6-Y-E-O-Z-E-N-0-P-H-C-2-H-U-J-1-2-6-J-A-L-1-E-B-9-U-Z-S-O-J-9-Z-[3]
- Initial crashing input sequence of the 3rd experiment:
I-9-N-T-W-G-R-E-8-2-8-Q-W-K-0-E-G-K-C-T-K-C-0-B-K-Y-M-M-G-D-U-W-F-C-6-[S]-G-K-9-7-2-9-7-N-E-T-2-4-J-H-2-J-N-1-K-G-T-U-T-0-9-U-J-P-5-W-Y-6-U-3-4-2-G-3-7-W-0-J-B-7-H-D-L-O-N-L-N-V-A-B-E-S-H-S-A-5-3-9-U-I-8-A-M-H-V-8-Z-M-E-K-M-R-P-C-T-D-S-X-J-[Q]-S-J-N-K-H-4-P-0-E-3-P-P-9-X-G-S-[R]-F-X-R-C-G-D-Y-O-N-0-G-I-K-1-9-U-B-0-4-N-P-M-O-M-S-H-G-Y-V-B-8-[T]-U-6-2-3-O-Y-B-6-6-J-L-Q-X-P-S-1-5-Y-O-G-K-E-G-J-U-J-1-J-B-X-2-M-[9]-A-E-M-G-E-I-O-6-F-C-3-O-R-E-M-Q-W-7-6-6-T-G-E-T-M-Z-6-5-Y-3-H-F-2-P-3-2-3-F-Q-4-7-B-R-I-L-[9]-5-4-[S]-C-[U]-Z-3-M-1-8-V-K-[B]-F-T-9-7-3-M-2-J-P-9-V-R-3-G-R-[1]-A-5-V-X-[0]-V-F-Z-F-O-X-1-O-P-M-7-B-3-R-J-C-J-1-D-V-Q-J-D-9-E-1-3-K-5-[0]
- Initial crashing input sequence of the 4th experiment:
K-2-Z-4-L-3-W-X-G-6-T-I-Z-8-M-Y-F-J-A-9-7-S-Q-R-G-K-H-E-[5]-Z-E-2-Q-7-Z-6-4-U-U-5-D-Y-2-X-G-9-8-3-9-K-[D]-6-2-7-9-X-5-W-Q-5-9-9-O-1-N-G-X-A-C-1-Q-1-Y-8-G-6-F-T-H-M-1-M-8-Z-C-G-C-A-A-M-9-F-4-0-9-M-F-K-9-N-6-3-C-Z-3-Y-[I]-L-Y-B-6-C-6-C-F-J-3-4-8-I-G-M-Y-J-T-A-7-G-Q-Y-O-8-R-6-2-W-5-8-Q-5-I-O-N-O-K-7-2-6-B-B-4-Y-9-J-O-0-T-H-[V]-T-C-I-L-X-W-A-Q-H-L-A-F-U-U-4-B-F-E-D-O-A-D-D-N-O-U-3-S-F-C-T-G-H-P-W-G-5-I-5-K-0-Y-M-7-A-S-Y-N-1-4-L-5-7-E-P-T-N-G-G-A-P-T-T-Z-L-D-5-C-J-F-T-A-[8]-0-D-K-C-I-C-E-1-A-J-X-A-M-G-F-9-O-R-P-J-4-W-B-W-C-C-I-C-H-J-M-N-V-7-8-C-D-Y-V-9-G-4-N-5-A-1-E-O-F-P-7-A-X-U-9-6-U-R-6-7-I-4-H-V-R-9-T-O-H-T-[S]-T-8-Q-5-Y-S-4-6-4-M-9-1-R-1-B-D-E-V-M-A-T-E-M-R-V-T-0-X-W-1-X-9-M-Z-0-9-R-O-P-3-H-O-C-7-I-3-G-O-5-1-W-V-E-5-V-L-Z-5-C-V-N-Y-3-B-R-C-B-8-X-S-4-L-C-Y-D-9-3-E-J-E-Q-F-M-Y-I-7-5-8-E-6-O-4-I-Q-D-9-O-S-K-I-F-4-Y-7-8-S-H-B-P-0-T-[U]-S-U-F-1-H-X-J-V-G-I-1-4-H-A-8-Z-A-3-E-[B]-B-0-P-O-P-E-0-1-5-8-[6]-W-[A]-P-S-H-E-Y-O-N-P-3-E-B-G-3-8-Y-W-8-M-N-3-G-U-G-G-B-L-N-S-A-F-9-L-X-C-1-M-I-7-A-6-5-4-S-U-4-M-U-9-8-G-M-8-P-5-8-K-F-S-P-R-7-6-C-V-Y-S-J-1-[D]-O-K-Y-L-U-F-0-9-8-T-[D]-J-7-S-Q-B-7-A-L-8-Q-Y-5-D-4-L-B-W-N-E-Z-Z-I-N-X-7-9-4-M-U-S-A-J-0-S-0-C-Q-U-M-N-8-Z-0-0-Q-A-D-Q-M-M-E-7-4-N-D-8-A-O-F-S-R-O-Q-I-S-I-S-K-R-V-M-Z-4-6-T-X-P-Y-3-M-X-8-E-M-P-1-A-Q-Z-8-G-T-X-M-R-I-[2]
- Initial crashing input sequence of the 5th experiment.

3-E-K-W-8-D-F-E-N-Y-9-J-J-S-8-H-W-L-W-E-P-D-7-[S]-H-Q-H-G-L-F-0-7-2-J-Y-C-Z-O-6-J-F-D-E-J-C-Q-3-6-4-X-3-O-5-U-B-S-1-U-G-C-S-Z-0-0-B-K-6-L-3-D-5-[I]-M-W-D-M-9-Q-4-K-D-2-H-G-H-0-U-5-B-L-I-0-X-U-5-C-1-[N]-R-2-I-D-A-2-U-J-R-E-0-K-J-Z-B-4-9-R-A-U-C-F-M-V-7-[P]-F-P-Z-J-M-O-D-U-G-J-5-Q-X-Y-M-2-Y-B-M-W-2-5-F-C-5-[I]-A-7-H-5-S-I-I-7-E-E-4-Z-J-Z-P-L-J-C-X-C-N-Q-P-F-5-H-1-Q-M-5-E-H-A-C-K-X-I-1-9-K-8-B-[D]-6-N-Z-0-L-L-V-8-8-J-7-Z-V-A-E-1-D-U-M-H-J-F-C-X-Y-3-J-A-U-Z-6-Q-F-7-E-N-7-K-H-0-6-1-V-9-D-7-L-K-0-3-E-6-H-N-N-N-Z-M-0-W-7-Y-6-Z-0-0-O-[I]-M-2-N-C-X-D-X-F-J-U-N-6-I-B-P-A-5-3-C-F-O-K-W-A-O-S-8-Q-S-E-N-D-6-K-A-B-9-H-7-L-R-E-D-B-Z-4-7-F-0-Q-W-I-J-4-9-P-5-G-6-P-1-E-P-6-5-G-G-F-F-Z-6-A-9-K-Q-C-U-B-J-N-Z-X-8-K-1-Y-K-6-C-P-E-R-H-K-C-D-U-U-C-C-1-B-1-7-9-N-R-S-E-3-B-0-C-1-2-9-W-Z-N-0-5-Y-P-D-F-G-B-0-3-U-Y-U-2-U-L-5-H-D-Q-F-Q-F-L-7-4-O-K-5-W-C-R-Y-[V]-S-K-7-Y-Q-E-4-Z-P-S-1-Y-J-W-E-L-O-4-9-I-J-U-9-Z-X-C-N-B-[2]-4-W-8-S-V-L-O-8-U-E-0-4-R-E-3-K-Q-9-3-U-3-K-S-P-6-Z-F-X-[D]-M-8-U-Z-F-8-G-R-G-5-R-B-U-Y-Z-R-V-8-5-C-2-C-G-3-L-U-M-Y-L-U-4-Q-V-B-M-Z-G-N-K-8-7-3-1-3-M-Z-5-P-U-6-U-E-P-4-C-C-J-H-V-A-X-W-7-[I]-1-C-Q-0-E-N-U-6-P-8-P-K-A-7-5-F-K-K-M-J-0-O-0-R-I-3-I-A-Y-2-0-5-N-Y-B-0-W-R-D-Q-7-0-1-0-0-Y-Z-4-L-J-[V]-S-F-N-4-1-4-Y-8-S-X-1-U-8-8-P-Q-3-H-7-N-B-Y-L-V-X-2-G-[C]-E-E-Y-F-1-5-R-Q-3-J-E-J-X-G-9-Z-B-U-6-R-Y-4-[O]-8-[S]-X-C-7-H-2-Q-I-3-5-M-D-N-D-F-C-F-N-E-3-H-M-I-O-Y-R-5-C-[P]-7-W-1-R-Z-Z-8-U-N-J-0-2-7-9-V-S-6-R-L-V-W-R-[I]-L-7-0-4-M-6-A-7-5-B-Y-Q-S-3-6-Z-Q-7-F-0-S-8-G-0-[D]-S-3-U-P-A-6-N-8-V-R-Y-L-N-M-A-6-B-E-Q-4-X-W-V-J-F-L-O-S-O-7-M-8-D-L-O-R-A-D-1-9-Z-A-P-0-D-[I]-4-Z-E-1-1-P-Y-E-W-C-J-[V]-G-Q-0-V-6-0-B-S-G-B-T-J-Y-3-9-X-K-W-K-3-P-T-X-H-4-7-M-9-Y-E-X-M-J-0-W-U-Q-A-G-X-D-X-5-C-U-U-L-P-5-D-I-R-T-9-Q-0-P-8-I-W-8-Z-3-U-M-X-L-D-0-9-L-Z-F-I-7-9-0-6-1-K-Z-N-8-6-E-7-9-N-S-G-J-M-8-F-R-5-S-L-4-K-8-B-3-Q-1-C-O-P-[2]

In each of these experiments, Recursive Delta Debugging, RDD, can find the first crashing sequence, which is included in the initial crash sequence of the experiment. The items of the crashing sequence which can be found by RDD algorithm in the initial sequence are marked by box-bordering. But RDD is incapable of finding the second crash sequence. In fact, the second crashing sequence of an experiment is shorter than the first crashing sequence of that experiment. However, RDD cannot simplify the initial sequence of an experiment to the second sequence of that experiment because the initial sequence does not contain the second crashing sequence. Alternatively, we use RL algorithms to explore new sequences. For instance, in the fifth experiment, the RDD algorithm can simplify the initial sequence to the sequence of "S-I-N-P-I-D-I-V-2-D-I-V-C-O-S-P-I-D-I-V-2". However, there exists a shorter crashing sequence of "T-A-N-P-I-D-I-V-2". CDM succeeds reaching the sequence of "T-A-N-P-I-D-I-V-2" using RL methods.

In the experiments, CDM simplifies the initial sequence using the RDD algorithm, then the RL algorithm of CDM uses this sequence to create initial states. After the initialization process, the RL agent explores new sequences and, for each experiment given, learns the second (shorter) crash sequence given in Table 4.

Table 5 shows the average elapsed time during the tests and gain with DGC. Each experiment set was repeated 100 times. From Table 5 we learn that with the use of DGC, the time to convergence decreases sharply when the problem has one optimal solution. Average time savings in terms of gains are 97% in SARSA tests and 94% in PS tests as illustrated in the Table 5.

Crash sequence length is a distinctive factor for the RL algorithm performance if other parameters of the environment are kept constant. As seen in Table 5, the performance of an algorithm is not always decreased with respect to the length of crash sequence due to other reasons, such as the distribution of actions on environment states, replicating actions on the crash scenario, and so on. The state where the action is executed can affect the performance of the RL algorithm. Thus, crash scenarios with smaller length of sequences are likely to take more time than crash scenarios with bigger length of sequences. For instance, the length of the shortest crash scenario which is "S-Q-R-T-N-E-G-1" in the 3rd experiment is smaller than that of "5-D-I-V-2-S-U-B-2" in the 4th experiment. On the other hand, as it is seen in Table 10, average elapsed time of the 3rd experiment is much higher than average elapsed time of the 4th experiment.

**Table 5**
Average Elapsed Time in Virtual Sequence Environment (in seconds).

| TN | Av.Time NoDGC SARSA | Av.Time withDGC SARSA | Gain % SARSA | Av.Time NoDGC PS | Av.Time withDGC PS | Gain % PS |
|---|---|---|---|---|---|---|
| 1 | 67.826 | **0.262** | 99 | 0.385 | **0.032** | 92 |
| 2 | 9.871 | **0.608** | 94 | 2.103 | **0.047** | 98 |
| 3 | 94.681 | **1.092** | 99 | 5.492 | **0.135** | 98 |
| 4 | 43.916 | **1.441** | 97 | 15.073 | **0.330** | 98 |
| 5 | 38.448 | **1.990** | 95 | 9.314 | **1.562** | 83 |
| | | | [a]**97** | | | [a]**94** |

[a]**Column Descriptions**:
**TN** : Test Number; **Av.Time NoDGC** : Average Elapsed Time (secs) of 100 tests without DGC, **Av.Time withDGC** : Average Elapsed Time (secs) of 100 tests with DGC, **Gain %** : Time Savings when DGC is activated.

**Table 6**
POS keys and functionalities.

| Key(s) | Description |
|---|---|
| [0–9] | There are 10 numeric keys to enter amount and to select menu item |
| E | Enter key is used to approve operation. |
| C | Clear key is used to clear last entered input. |
| X | Escape key is used to exit from menu. |
| ♯ | Sharp or pound key is used to open menu |
| * | Star is used as a function key. |

### 5.2.2. Test with POS environment

This environment supports four main POS operations, namely *sale*, *void*, *refund*, and *batch close* operations. The *sale* operation is applied when a customer pays for the goods. The *void* operation is a requirement for cashback when a product is returned on the same day of the original transaction. The *refund* operation is applied when a customer wants to return a product purchased in the previous days. The *batch close* operation is required for reconciliation with the bank. Normally, the merchant closes the batch at the end of the day. The *sale* operation is simply performed by entering an amount and pressing the Enter key on the idle screen. For other operations users first enter menu using the ♯ key.

In the experiments, the POS device was not connected to any bank host and no credit card was requested. Instead, offline POS approved all requested transactions, allowing CDM to explore bugs in the SUT's operations without waiting for external interactions. Fifteen keys were located on the keyboard of the POS device, as shown in Fig. 5. The functionalities of the keys were described in Table 6.

In this section, we explained the flow of the CDM on simplification of a bug scenario that the bug causes the SUT to crash. In the initiation phase of the experiments. CDM sends randomly generated inputs to a POS application one by one until the POS application crashes. When CDM receives no response from the POS application to a request containing an input to be executed, CDM decides that the POS application has crashed and the generated inputs are used as the initial crashing sequence. After the initialization phase, CDM tries to simplify the obtained crashing sequence. For this purpose, CDM first applies the RDD algorithm to remove bug-unrelated inputs from the

**Table 7**
The optimum solutions for the POS crash problem. There are 84 optimum solutions.

| Sale Amn. | Refund Amn.1 | Refund Amn.2 | Whole Sequence | ♯ Opt Solns. |
|---|---|---|---|---|
| 3 | 2 | 2 | 3 - E - ♯ - 3 - ♯ - 2 - 2 - E - 2 - E - ♯ - 3 | 1 |
| 4 | 2 | 3 | 4 - E - ♯ - 2 - ♯ - 3 - 2 - E - 2 - E - ♯ - 3 | 3 |
| | 3 | 2 | 4 - E - ♯ - 3 - ♯ - 2 - 2 - E - 2 - E - ♯ - 3 | |
| | 3 | 3 | 4 - E - ♯ - 3 - ♯ - 3 - 2 - E - 2 - E - ♯ - 3 | |
| 5 | 2 | 4 | 5 - E - ♯ - 2 - ♯ - 4 - 2 - E - 2 - E - ♯ - 3 | 6 |
| | 4 | 2 | 5 - E - ♯ - 4 - ♯ - 2 - 2 - E - 2 - E - ♯ - 3 | |
| | ... | ... | ..... | |
| | 4 | 4 | 5 - E - ♯ - 4 - ♯ - 4 - 2 - E - 2 - E - ♯ - 3 | |
| 6 | 2 | 5 | 6 - E - ♯ - 2 - ♯ - 5 - 2 - E - 2 - E - ♯ - 3 | 10 |
| | 5 | 2 | 6 - E - ♯ - 5 - ♯ - 2 - 2 - E - 2 - E - ♯ - 3 | |
| | ... | ... | ..... | |
| | 5 | 5 | 6 - E - ♯ - 5 - ♯ - 5 - 2 - E - 2 - E - ♯ - 3 | |
| 7 | 2 | 6 | 7 - E - ♯ - 2 - ♯ - 6 - 2 - E - 2 - E - ♯ - 3 | 15 |
| | 6 | 2 | 7 - E - ♯ - 6 - ♯ - 2 - 2 - E - 2 - E - ♯ - 3 | |
| | ... | ... | ..... | |
| | 6 | 6 | 7 - E - ♯ - 6 - ♯ - 6 - 2 - E - 2 - E - ♯ - 3 | |
| 8 | 2 | 7 | 8 - E - ♯ - 2 - ♯ - 7 - 2 - E - 2 - E - ♯ - 3 | 21 |
| | 7 | 2 | 8 - E - ♯ - 7 - ♯ - 2 - 2 - E - 2 - E - ♯ - 3 | |
| | ... | ... | ..... | |
| | 7 | 7 | 8 - E - ♯ - 7 - ♯ - 7 - 2 - E - 2 - E - ♯ - 3 | |
| 9 | 2 | 8 | 9 - E - ♯ - 2 - ♯ - 8 - 2 - E - 2 - E - ♯ - 3 | 28 |
| | 8 | 2 | 9 - E - ♯ - 8 - ♯ - 2 - 2 - E - 2 - E - ♯ - 3 | |
| | ... | ... | ..... | |
| | 8 | 8 | 9 - E - ♯ - 8 - ♯ - 8 - 2 - E - 2 - E - ♯ - 3 | |

initial crashing sequence. Then, CDM uses RL algorithms to find shorter crash scenarios.

The POS application in this experiment has a bug as follows:

- The total sale amount in the first batch is A1 and the batch is closed.
- The second batch contains partial refunds whose amounts are PR1 and PR2.
- On closing the second batch, if the total of PR1 and PR2 is bigger than A1, the application flow enters an endless loop. If PR1 and PR2 were done separately (by exiting the refund menu), no bugs occurred. Because the total amount of the previous batch and that of the current refund are calculated when the refund menu is selected. The bug reason is that the total current refund amount is not updated during the partial refund.

.

The POS crash problem has multiple optimum solutions where optimum value is 12. Number of refund combinations increases in a triangular series logic when the sale amount in the first batch increases. The number of optimum solutions for the sale amounts of 3, 4, 5, 6, 7, 8, 9 in the first batch are 1, 3, 6, 10, 15, 21, 28, respectively. Table 7 shows the refund combinations for each sale.

We explained details of three of the optimum scenarios in Table 8. In the first row, since the total of the refund amounts in the second batch had exceeded the positive amount in the first batch, the POS application freezes during the second batch close operation. However, testers verified that the POS application did not allow the refund operation when the refund was tried with the amount of 1.00. The POS application checked whether the refund amount was bigger than the previous batch amount or not on entering the refund menu. In the problem case, the refund amount was divided into parts and each amount did not exceed the amount of the first batch. For this reason, a negative total occurred in the second batch and the POS application crashed.

Table 9 contains the length of initial crash sequences and corresponding crashing sequence simplified by the RDD algorithm. At the end of learning, the RL agent can find the sequences with length of 12 which is smaller than lengths of sequences in Table 9.

**Table 8**
Three samples for shortest scenarios. In this experiment, the shortest crashing sequence length is 12.

| CN | Inputs | Description |
|---|---|---|
| 01 | 8 - E | Sale with amount of 0.08 |
| | ♯ - 3 | Batch close operation |
| | ♯ - 2 | Enter Refund Menu |
| | 5 - E | Refund with amount of 0.05 (<0.08) |
| | 5 - E | Refund with amount of 0.05 (<0.08) |
| | ♯ - 3 | Batch close operation |
| 02 | 7 - E | Sale with amount of 0.07 |
| | ♯ - 3 | Batch close operation |
| | ♯ - 2 | Enter Refund Menu |
| | 2 - E | Refund with amount of 0.02 (<0.07) |
| | 6 - E | Refund with amount of 0.06 (<0.07) |
| | ♯ - 3 | Batch close operation |
| 03 | 9 - E | Sale with amount of 0.09 |
| | ♯ - 3 | Batch close operation |
| | ♯ - 2 | Enter Refund Menu |
| | 8 - E | Refund with amount of 0.08 (<0.09) |
| | 4 - E | Refund with amount of 0.04 (<0.09) |
| | ♯ - 3 | Batch close operation |

According to the results of experiments with the POS environment in Table 10, DGC contributes to simplifying crashing sequences. Each experiment set was run 100 times. Having multiple optimum solutions delayed the DGC algorithm to start in PS tests. Thus, the performance contribution of DGC in this experiment was less compared to that in the experiments where the problem has a single optimum solution. However, the contribution of DGC in saving time is undeniable. Average time savings in terms of gains are 91% in SARSA tests and 24% in PS tests.

#### 5.2.3. Comparison with RandRDD

In this section, we compared the performance of the CDM with the RandRDD module to demonstrate the RL contribution on exploring new crashing scenarios. In this experiment, we used the POS environment described in Section 5.2.2 where the length of the optimum crashing sequence is 12. The results show CDM outperformed the RandRDD module which combines random input generation with RDD. RandRDD has the ability to discover new crash sequences thanks to RandRDD's random input generation functionality. The RDD contribution to this combination is the simplification of the randomly found crashing sequence.

In a RandRDD episode, first, individual randomly generated inputs were sent to the SUT to cause the SUT to crash. Next, the RDD algorithm simplified the crashing sequence. A RandRDD test section consisted of 9 episodes, the shortest crashing sequence among them was selected. We empirically obtained the number of 9. The best of 9 episodes was close to the optimum crashing sequence length that was 12. (See Table 11).

The average values of the time elapsed in experiments with RandRDD, PS and PS with DGC have been presented in Table 12. We performed tests on the SUT with 5 configurations for input execution time. PS outperformed RandRDD by 16% and DGC improved this PS performance by up to 44%. These results convince us to use e-greedy selection based algorithms to find the shortest crash sequences of a given problem.

We statistically tested our results to check if they were significantly different. We used Welch's t-test which is applied when variances of two samples are unequal. We divided the output data of each experiment into two parts to obtain two samples. Then we run Welch's t-test using these samples. Results in Table 13 show that all experimental results belonging to PS (both DGC is active and passive) are significantly not different. Thus, we conclude that time savings observed in PS experiments were not obtained in a chance.

To summarize, we submitted the experimental setup that explains how RL can be applied to find minimum crashing sequences. We

**Table 9**
Sequences found by RDD. During the tests, long crash sequences are first simplified by RDD. RL algorithms find sequences shorter than these sequences.

| TN | Initial length | Crash Sequences Found by RDD |
|---|---|---|
| 01 | 1675 | E - 9 - E - ♯ - 3 - ♯ - 2 - 6 - E - 6 - E - ♯ - 3 |
| 02 | 6744 | 5 - 1 - E - ♯ - 3 - ♯ - 2 - 4 - 7 - E - 9 - E - ♯ - 3 |
| 03 | 2364 | E - 6 - 9 - E - ♯ - 3 - ♯ - 2 - 6 - 6 - E - 4 - E - ♯ - 3 |
| 04 | 3505 | ♯ - 3 - 5 - E - ♯ - 3 - ♯ - 2 - 3 - E - 2 - E - 1 - E - ♯ - 3 |
| 05 | 1714 | 2 - 2 - E - ♯ - 3 - ♯ - 2 - 7 - E - 9 - E - ♯ - 2 - 9 - E - ♯ - 3 |
| 06 | 5278 | ♯ - 3 - 5 - E - ♯ - 3 - ♯ - 2 - 3 - E - 1 - ♯ - 2 - 3 - E - 3 - ♯ - 3 |
| 07 | 212 | 7 - 3 - E - 4 - 7 - E - ♯ - 3 - ♯ - 2 - 9 - 8 - E - 6 - 1 - E - 7 - ♯ - 3 |
| 08 | 6409 | 1 - 1 - E - ♯ - 3 - 5 - 3 - E - ♯ - 3 - ♯ - 2 - 1 - 5 - E - 4 - 8 - E - 1 - ♯ - 3 |
| 09 | 3118 | ♯ - ♯ - X - 3 - 4 - E - ♯ - 1 - 5 - ♯ - 3 - ♯ - 2 - 8 - E - 1 - 4 - E - 2 - 3 - E - ♯ - 3 |
| 10 | 4852 | 1 - 7 - 2 - E - 7 - 0 - E - 5 - 1 - 0 - 2 - E - ♯ - 3 - ♯ - 2 - 6 - 5 - 9 - E - 4 - 8 - 0 - 2 - E - ♯ - 3 |

**Table 10**
Average Elapsed Time in POS Environment (in seconds).

| TN | Av.Time NoDGC SARSA | Av.Time withDGC SARSA | Gain % SARSA | Av.Time NoDGC PS | Av.Time withDGC PS | Gain % PS |
|---|---|---|---|---|---|---|
| 01 | 4.595 | **0.372** | 92 | 0.167 | **0.139** | 17 |
| 02 | 5.488 | **0.346** | 94 | 0.106 | **0.091** | 14 |
| 03 | 5.739 | **0.379** | 93 | 0.138 | **0.093** | 33 |
| 04 | 4.898 | **0.358** | 93 | 0.135 | **0.116** | 14 |
| 05 | 3.815 | **0.461** | 88 | 0.141 | **0.106** | 25 |
| 06 | 4.055 | **0.341** | 92 | 0.117 | **0.088** | 25 |
| 07 | 5.512 | **0.452** | 92 | 0.503 | **0.299** | 41 |
| 08 | 4.986 | **0.368** | 93 | 0.268 | **0.191** | 29 |
| 09 | 5.281 | **0.460** | 91 | 0.274 | **0.194** | 29 |
| 10 | 6.535 | **0.518** | 92 | 0.422 | **0.332** | 21 |
| | | | [a]**91** | | | [a]**24** |

[a]**Column Descriptions**:
**TN** : Test Number; **Av.Time NoDGC** : Average Elapsed Time(secs) of 100 tests without DGC, **Av.Time withDGC** : Average Elapsed Time(secs) of 100 tests with DGC, **Gain %** : Time Savings when DGC is activated.

**Table 11**
Averages of the shortest sequence lengths. RandGen episode count is set to 9, because the error rate of 0.005 is acceptable.

| TN | Repeat count | Average length | Error wrt 12 |
|---|---|---|---|
| 1 | 1 | 14.96 | 24.67% |
| 2 | 5 | 12.32 | 2.67% |
| 3 | 8 | 12.10 | 0.83% |
| 4 | 9 | 12.06 | 0.50% |

**Table 12**
PS performs better than RandRDD on obtaining the shortest crash sequence.

| Key E.T.(ms) | Avg.Time RRDD | Avg.Time PS | Avg.Time PSwDGC | Gain (%) PS/RRDD | Gain (%) PSwDGC/RRDD |
|---|---|---|---|---|---|
| 1 | 274.590 | 237.732 | 173.130 | 13 | 37 |
| 2 | 467.129 | 411.241 | 246.756 | 12 | 47 |
| 3 | 690.137 | 555.167 | 330.336 | 19 | 52 |
| 4 | 856.699 | 699.769 | 459.983 | 18 | 46 |
| 5 | 1043.099 | 840.429 | 636.806 | 19 | 39 |
| | | | | [a]**16** | [a]**44** |

[a]**Column Descriptions**:
**Key E.T.(ms)** : Required time for SUT to execute 1 input/entry; **Av.Time RRDD** : Average Elapsed Time(secs) of 100 tests in Random RDD test, **Av.Time PS** : Average Elapsed Time(secs) of 100 tests in PS (DGC is not activated), **Av.Time PSwDGC** : Average Elapsed Time(secs) of 100 tests in PS when DGC is activated, **Gain % PS/RRDD** : Time Savings of PS over Random RDD. **Gain % PSwDGC/RRDD** : Time Savings of DGC activated PS over Random RDD.

emphasize that the algorithms used in CDM do not depend on what the values in the input set and crash sequence scenarios are. In other words, we applied an auto blackbox test on the SUT. In this section, we also discussed the test results where DGC makes CDM faster to find the shortest crashing sequence.

**Table 13**
Statistical Test where $\alpha$ is set to 0.05. Only a test with 3ms of RandRDD failed. Others are not significantly different.

| Key E.T. (ms) | $P(T \leq t)$ RRDD | $P(T \leq t)$ PS | $P(T \leq t)$ PSwDGC |
|---|---|---|---|
| 1 | 0.332 | 0.474 | 0.309 |
| 2 | 0.196 | 0.947 | 0.388 |
| 3 | **0.026** | 0.449 | 0.861 |
| 4 | 0.854 | 0.731 | 0.570 |
| 5 | 0.979 | 0.808 | 0.706 |

*__Column Descriptions__:
**Key E.T.(ms)** : Required time for SUT to execute 1 input/entry; $P(T \leq t)$ **RRDD** : Two-tail Test of Result of Random RDD, $P(T \leq t)$ **PS** : Two-tailTest of Result of PS, (DGC is not activated), $P(T \leq t)$ **PSwDGC** : Two-tail Test of Result of PS when DGC is activated.

### 5.3. Challenges and validity constraints

We share a brief discussion relating to the field challenges of this work. We prepared *CDM* as a library which can be configured by external applications. *CDM* would be helpful for both software developers and QA teams because it obtains a simple crashing sequence. However, the *CDM* integration process faces difficulties. A case of these is the incomplete input set of the SUT. For instance, we see that a flow in the SUT is triggered periodically and this periodic call is not involved in the input set. However, we should not ignore triggering that flow during the test. We tackle these missing input lists by a well-documented checklist, which contains possible program inputs. The next challenge is the decision for the state where the SUT is unresponsive. *CDM* should be able to determine whether the SUT is in a blocking state or the SUT is awaiting another process/server. To overcome this challenge, the SUT is modified to set a flag when it waits for another process. *CDM* checks this flag to avoid a wrong crash decision. When the flag is set, *CDM* waits longer for the action response. Another challenge during the integration is that SUT is impossible for modification. This case occurs when an application does not have a TCP/COM listener to wait for inputs, and we cannot change its code to add a listener. We attain a solution for this difficulty with a bridge application. For example, if SUT runs on Windows, *CDM* sends generated inputs to a bridge application, which has a TCP listener. And the bridge application using "user32.dll" makes system calls to create key press events for Windows applications.

We want to draw a frame of constraints where CDM obtains successful results. The first validity constraint of the method proposed in this study is that all inputs must be generated by CDM during the test of SUT. We do not allow other interfaces of SUT to produce any input. Another constraint is that we should be able to programmatically restart the SUT. In high-level systems, if the SUT become unresponsive, CDM sends a signal to kill the process and starts a new instance of the SUT. In embedded systems, if an application blocks the system, the device is restarted. If the embedded system on which the SUT runs does not support auto restart, we should set up a watchdog timer to restart the device when the SUT turns into a blocking state. Another requirement for the system is a feature of restoring such as reset to

factory settings. When the agent of RL in the CDM reaches the goal state, new episode should start at initial environmental settings. To satisfy this constraint, CDM applies backup operation at the beginning of the test. And at the beginning of each episode, and CDM restores the resources used by the SUT.

## 6. Conclusion and future work

In this study, we defined the software crash problem as a RL problem and we employed SARSA and PS to learn the shortest crash sequence. Proposed module, *CDM*, first simplifies the longer crashing sequence using delta debugging techniques, then reinforcement learning algorithms explore shorter crashing sequences related to the same bug. In addition, we presented DGC to speed up the convergence to the optimal path. Performance contribution of DGC to both SARSA and PS is thoroughly analyzed and demonstrated in this paper. The results show that RL algorithms with DGC find crashing input sequences faster than RL algorithms without DGC.

Our RL approach can also detect crash sequences of applications running on real systems. For this purpose, the proposed black-box test modules with RL algorithms may serve as part of a real application as a library. Here, the agent in the RL library generates the inputs of the application to explore the shortest crash sequence. Another option for integrating RL modules with real applications is TCP (Transport Control Protocol) or serial port (RS232 or USB Serial). In this option, the agent produces actions and sends them over to one of the communication channels.

We would also like to share our answers to the question of how DGC copes with multiple crash scenarios in an application. First, DGC is not activated at the beginning episodes, it is activated when the variance of the latest explored crash sequences is smaller than the threshold value calculated by the logarithmic function in Eq. (3). When such episodes have been reached, the agent would be around only one crash scenario. In this case there are multiple scenarios in the range of a small threshold, the crash scenario of shortest length would be chosen. After developers fix the bug, *CDM* runs again to find other potential scenarios.

In future work, we would like to show that RL algorithms with DGC can also handle sequences in oracle problems that bugs do not necessarily lead to crashes. In the crash problem discussed in this article, the agent recognizes the goal state because crash is a special state in which the system becomes unresponsive. The difficulty in finding bugs that do not cause crashes is that the agent cannot recognize the goal state. We will also study extending the scope of DGC, which accelerates RL algorithms. In other words, we will study the integration of DGC in search of solving problems other than crash problems with RL algorithms.

## CRediT authorship contribution statement

**Engin Durmaz:** Conceptualization, Methodology, Software, Data curation, Validation, Writing – original draft, Visualization, Investigation, Writing – review & editing, Resources, Formal analysis. **M. Borahan Tümer:** Supervision, Methodology, Formal analysis, Writing – review & editing.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References


Adamo, D., Khan, M. K., Koppula, S., & Bryce, R. (2018). Reinforcement learning for android gui testing. In *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation* (pp. 2–8). New York, NY, USA: Association for Computing Machinery.

Ami, A. S., Hasan, M. M., Rahman, M. R., & Sakib, K. (2018). Mobicomonkey: context testing of android apps. In *Proceedings of the 5th international conference on mobile software engineering and systems* (pp. 76–79). New York, NY, USA: Association for Computing Machinery.

Arcuri, A. (2019). Restful api automated test case generation with evomaster. *ACM Transactions on Software Engineering and Methodology*, *28*, 1–37.

Arnatovich, Y., Wang, L., Ngo, N., & Soh, C. (2018). Mobolic: an automated approach to exercising mobile application guis using symbiosis of online testing technique and customated input generation. *Software: Practice and Experience*, *48*.

Chen, L., & Dey, S. (2001). Software-based self-testing methodology for processor cores. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *20*(3), 369–380.

Chen, N., & Kim, S. (2015). Star: stack trace based automatic crash reproduction via symbolic execution. *IEEE Transactions on Software Engineering*, *41*(2), 198–220.

Choi, K., Ko, M., & Chang, B.-m. (2018). A practical intent fuzzing tool for robustness of inter-component communication in android apps. *KSII Transactions on Internet and Information Systems*, *12*, 4248–4270.

Derakhshanfar, P., Devroey, X., Panichella, A., Zaidman, A., & van Deursen, A. (2020). Botsing, a search-based crash reproduction framework for Java. In *2020 35th IEEE/ACM international conference on automated software engineering* (pp. 1278–1282).

Fevotte, F. c., & Lathuilière, B. (2019). Debugging and optimization of HPC programs with the Verrou tool. In *2019 IEEE/ACM 3rd international workshop on software correctness for HPC applications (Correctness)* (pp. 1–10).

Fraser, G., & Arcuri, A. (2011). Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering* (pp. 416–419). New York, NY, USA: Association for Computing Machinery.

Hutchison, C., Zizyte, M., Lanigan, P. E., Guttendorf, D., Wagner, M., Le Goues, C., & Koopman, P. (2018). Robustness testing of autonomy software. In *2018 IEEE/ACM 40th international conference on software engineering: Software engineering in practice track* (pp. 276–285).

Ispoglou, K. K., Austin, D., Mohan, V., & Payer, M. (2020). FuzzGen: Automatic fuzzer generation. In *USENIX security symposium*.

Kim, J., Kwon, M., & Yoo, S. (2018). Generating test input with deep reinforcement learning. In *2018 IEEE/ACM 11th international workshop on search-based software testing* (pp. 51–58).

Kiss, A., Hodován, R., & Gyimóthy, T. (2018). HDDr: A recursive variant of the hierarchical delta debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation* (pp. 16–22). New York, NY, USA: Association for Computing Machinery.

Koroglu, Y., Sen, A., Muslu, O., Mete, Y., Ulker, C., Tanriverdi, T., & Donmez, Y. (2018). QBE: QLearning-based exploration of Android applications. In *2018 IEEE 11th international conference on software testing, verification and validation* (pp. 105–115).

Lawanna, A. (2012). The theory of software testing. *Intelligent Transportation Systems Journal*, *16*(1), 35–40.

Lenarduzzi, V., Stan, A. C., Taibi, D., Venters, G., & Windegger, M. (2018). Prioritizing corrective maintenance activities for android applications: an industrial case study on android crash reports. In D. Winkler, S. Biffl, & J. Bergsmann (Eds.), *Software quality: Methods and tools for better software and systems* (pp. 133–143). Vienna, Austria: Springer.

Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., & Zhai, C. (2006). Have things changed now? An empirical study of bug characteristics in modern open source software. In J. Torrellas (Ed.), *ASID'06: 1st workshop on architectural and system support for improving software dependability* (pp. 25–33). San Jose, California, USA: ACM.

Liljedahl, F. (2019). *Exploring the possibilities of robustness testing of CoAP implementations using evolutionary fuzzing* (Master's thesis), School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology in Stockholm.

Mao, K., Harman, M., & Jia, Y. (2016). Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th international symposium on software testing and analysis* (pp. 94–105).

Moore, A. W., & Atkeson, C. G. (1993). Prioritized sweeping: reinforcement learning with less data and less time. *Machine Learning*, *13*(1), 103–130.

Moran, K., Linares, M., Bernal, C., Vendome, C., & Poshyvanyk, D. (2017). CrashScope: A practical tool for automated testing of Android applications. In *2017 IEEE/ACM 39th international conference on software engineering companion* (pp. 15–18).

Nagy, S., & Hicks, M. (2019). Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE symposium on security and privacy* (pp. 787–802).

Nayrolles, M., Hamou-Lhadj, A., Tahar, S., & Larsson, A. (2017). A bug reproduction approach based on directed model checking and crash traces. *Journal of Software: Evolution and Process*, *29*.

Orso, A., Joshi, S., Burger, M., & Zeller, A. (2006). Isolating relevant component interactions with JINSI. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis* (pp. 3–10). New York, NY, USA: ACM.

Panichella, A., Kifetew, F. M., & Tonella, P. (2018). Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, *44*(2), 122–158.

Pradel, M., & Sen, K. (2017). *Deep learning to find bugs*. TU Darmstadt, Germany: TU Darmstadt, Department of Computer Science.

Rawat, M. S., & Dubey, S. K. (2012). Software defect prediction models for quality improvement: A literature study. *International Journal of Computer Science Issues*, *9*(2), 288–296.

Riganelli, O., Mottadelli, S. P., Rota, C., Micucci, D., & Mariani, L. (2020). Data loss detector: Automatically revealing data loss bugs in android apps. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis* (pp. 141–152). New York, NY, USA: Association for Computing Machinery.

Röß ler, J., Zeller, A., Fraser, G., Zamfir, C., & Candea, G. (2013). Reconstructing core dumps. In *2013 IEEE sixth international conference on software testing, verification and validation* (pp. 114–123).

Ruwase, O., & Lam, M. S. (2004). A practical dynamic buffer overflow detector. In NDSS04 (Ed.), *11th annual network and distributed system security symposium* (pp. 159–169). San Diego, California, USA: The Internet Society.

Singh, D. S. K., & Singh, D. A. (2019). *Software testing* (1st ed). Vandana Publications Lucknow.

Soltani, M., Derakhshanfar, P., Devroey, X., & Deursen, A. (2020). A benchmark-based evaluation of search-based crash reproduction. *Empirical Software Engineering*, *25*.

Soltani, M., Panichella, A., & van Deursen, A. (2020). Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering*, *46*(12), 1294–1317.

Spieker, H., Gotlieb, A., Marijan, D., & Mossige, M. (2017). Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis* (p. 12–22).

Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: an introduction, (2nd ed). The MIT Press.

Traon, Y., Jéron, T., Jézéquel, J.-M., & Morel, P. (2000). Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability*, *49*(1), 12–25.

Vuong, T. A. T., & Takada, S. (2018). A reinforcement learning based approach to automated testing of android applications. In *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation* (pp. 31–37). New York, NY, USA: Association for Computing Machinery.

Wang, J., Chen, J., Sun, Y., Ma, X., Wang, D., Sun, J., & Cheng, P. (2021). Robot: Robustness-oriented testing for deep learning systems.

Wang, S., Gotlieb, A., Ali, S., & Liaaen, M. (2013). Automated test case selection using feature model: An industrial case study. In *Model-driven engineering languages and systems* (pp. 237 – 253).

Wei, H., Ruilian, Z., & Qunxiong, Z. (2015). Integrating evolutionary testing with reinforcement learning for automated test generation of object-oriented software integration. *Journal of Electronics*, *24*(1).

Xiao, L., Li, Y., Huang, X., & Du, X. (2017). Cloud-based malware detection game for mobile devices with offloading. *IEEE Transactions on Mobile Computing*, *16*(10), 2742–2750.

Xuan, J., Xie, X., & Monperrus, M. (2015). Crash reproduction via test case mutation: let existing test cases help. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (pp. 910–913). New York, NY, USA: Association for Computing Machinery.

Zeller, A. (2002). Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on foundations of software engineering* (pp. 1–10). New York, NY, USA: Association for Computing Machinery.

Zheng, Y., Liu, Y., Xie, X., Liu, Y., Ma, L., Hao, J., & Liu, Y. (2021). Automatic web testing using curiosity-driven reinforcement learning. In *2021 IEEE/ACM 43rd international conference on software engineering* (pp. 423–435).