Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

UNIVERSITÉ DE NAMUR

RESEARCH OUTPUTS / RÉSULTATS DE RECHERCHE

Search-based Crash Reproduction using Behavioral Model Seeding

Derakhshanfar, Pouria; Devroey, Xavier; Perrouin, Gilles; Zaidman, Andy; Deursen, Arie van

Published in:
ArXiv pre-print

Publication date:
2019

Link to publication

Citation for pulished version (HARVARD):
Derakhshanfar, P, Devroey, X, Perrouin, G, Zaidman, A & Deursen, AV 2019, 'Search-based Crash Reproduction using Behavioral Model Seeding', *ArXiv pre-print*.

Search-based Crash Reproduction using Behavioral Model Seeding

Pouria Derakhshanfar[1*], Xavier Devroey[1], Gilles Perrouin[2],
Andy Zaidman[1] and Arie van Deursen[1]

[1] *Delft University of Technology, Postbus 5, 2600 AA Delft, The Netherlands*
[2] *FNRS Research Associate, PReCISE, NADI, University of Namur, Rue de Bruxelles 61, 5000 Namur, Belgium*

SUMMARY

Search-based crash reproduction approaches assist developers during debugging by generating a test case which reproduces a crash given its stack trace. One of the fundamental steps of this approach is creating objects needed to trigger the crash. One way to overcome this limitation is seeding: using information about the application during the search process. With seeding, the existing usages of classes can be used in the search process to produce realistic sequences of method calls which create the required objects. In this study, we introduce behavioral model seeding: a new seeding method which learns class usages from both the system under test and existing test cases. Learned usages are then synthesized in a behavioral model (state machine). Then, this model serves to guide the evolutionary process. To assess behavioral model-seeding, we evaluate it against test-seeding (the state-of-the-art technique for seeding realistic objects) and no-seeding (without seeding any class usage). For this evaluation, we use a benchmark of 124 hard-to-reproduce crashes stemming from six open-source projects. Our results indicate that behavioral model-seeding outperforms both test seeding and no-seeding by a minimum of 6% without any notable negative impact on efficiency. Copyright © 2019 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: seed learning, crash reproduction, search-based software testing

1. INTRODUCTION

The starting point of any debugging activity is to try to reproduce the problem reported by a user in the development environment [1, 2]. In particular, for Java programs, when a crash occurs, an exception is thrown. A developer strives to reproduce it to understand its cause, then fix the bug, and finally add a (non-)regression test to avoid reintroducing the bug in future versions.

Manual crash reproduction can be a challenging and labor-intensive task for developers: it is often an iterative process that requires setting the debugging environment in a similar enough state as the

environment in which the crash occurred [1]. Moreover, it requires the developer to have knowledge of the system's components involved in the crash. To help developers in their task, several automated crash reproduction methods, relying on different techniques, have been proposed [3, 4, 5, 6, 7, 8, 9].

One of the most promising approaches is to generate crash-reproducing test cases using search-based software testing [8, 9]. This approach benefits from a *guided genetic algorithm* which searches for the crash reproducing test case. Soltani *et al.* performed an empirical evaluation of EvoCrash and reported that it outperforms other crash reproduction approaches [8].

One of the challenges of search-based crash reproduction is to bring enough information into the test generation process. For instance, complex elements (like strings with a particular format or objects with a complex structure) are hard to initialize without additional information. This can lead to two different issues: first, complex elements take more time to be generated, which can prevent to find a solution within the time budget allocated to the search; and second, elements that require complex initialization procedures (*e.g.,* specific sequences of method calls to set up an object) may prevent the search to start if the search-based approach is unable to create an initial population.

Rojas *et al.* [10] demonstrated that *seeding* is beneficial for search-based unit test generation. More specifically, by analyzing source code (collecting information that relates to numeric values, string values, and class types) and existing tests (collecting information about the behavior of the objects in the test) and making them available for the search process, the overall coverage of the generated test improves. However, current seeding strategies focus on collecting and reusing values and object states as-is.

In this paper, we define, implement, and evaluate a new seeding strategy, called *behavioral model seeding*, which abstracts behavior observed in the source code and test cases using transition systems. The transition systems represent the (observed) usages of the classes and are used during the search to generate objects and sequences of method calls on those objects.

Behavioral model seeding takes advantage of the advances made by the model-based testing community [11] and uses them to enhance search-based software testing. This seeding strategy helps the search process: (i) it provides the possibility of covering the given crash by collecting information from various resources (*e.g.,* source code and existing test cases) to infer a unique transition system; and (ii) it finds the most beneficial seeding candidates, for guiding the crash reproduction search process, by defining a rational procedure for the selection of abstract object behaviors from the inferred models.

We also adapt *test seeding*, introduced by Rojas *et al.*, for search-based crash reproduction. Contrarily to model seeding, *test seeding* relies only on the states of the objects observed during the execution of the test to seed a search process. Unlike search-based unit test generation, search-based crash reproduction does not seek to maximize the coverage of the class, but rather generates a specific test case able to reproduce a crash. Since test seeding has only been applied to search-based unit test generation [10], we first evaluate the use of test seeding for crash reproduction. We then compare the results of test seeding with the application of *model seeding*, which combines information on the objects states coming from the test cases with information collected in the source code, to search-based crash reproduction.

We performed an evaluation on 124 crashes from 6 open-source applications to answer the following research questions:

RQ1 What is the influence of *test seeding used during initialization* on search-based crash reproduction?

RQ2 What is the influence of *behavioral model seeding used during initialization* on search-based crash reproduction?

We consider both research questions from the perspective of *effectiveness* (of initializing the population and reproducing crashes) and *efficiency*. We also investigate the factors (*e.g.,* the cost of analyzing existing tests) that influence the test and model seeding approaches and gain a better insight into how search-based crash reproduction works and how it can be improved. Generally, our results indicate that behavioral model seeding increases the number of crashes that we can reproduce. More specifically, because of the randomness in the test generation process, we execute the crash replication multiple times and we observe that in the majority of these executions 4 crashes (out of 124) can be replicated; also, this seeding strategy can reproduce 9 crashes, which are not reproducible at all with no seeding, in at least one execution. In addition, this seeding strategy slightly improves the efficiency of the crash reproduction process. Moreover, model seeding enables the search process to start for three additional crashes. In contrast, using test seeding in crash-reproduction leads to a lower crash-reproduction rate and search initialization.

The contributions of this paper are:

1. An evaluation of test seeding techniques applied to search-based crash reproduction;
2. A novel behavioral model seeding approach for search-based software testing and its application to search-based crash reproduction;
3. An open source implementation of model seeding in the Botsing toolset[†]; and
4. The discussion of our results demonstrating improvements in search-based crash reproduction abilities and contributing to a better understanding of the search-based process. All our results are available in the replication package.

The remainder of the paper is structured as follows: Section 2 provides background on search-based crash reproduction, and model-based testing. Section 3 describes our behavioral model seeding strategy. Section 4 details our implementation, while Section 5 explains the evaluation setup. Section 6 presents our results. We discuss them and explain threats to our empirical analyses in Section 7. Section 8 discusses future work and Section 9 wraps up the paper.

2. BACKGROUND AND RELATED WORK

Application crashes that happen while the system is operating are usually reported to developer teams through an issue tracking system for debugging purposes [12]. Depending on the amount of information reported from the operations environment, this debugging process may take more or less time. Typically, the first step for the developer is to try to reproduce the crash in his development environment [1]. Various approaches [7, 3, 5, 8, 6] automate this process and generate a *crash-reproducing test case* without requiring human intervention during the generation process.

[†]Available at https://github.com/STAMP-project/botsing.

Listing 1: Stack trace of the XWIKI-13372 crash

```
0  java.lang.NullPointerException: null
1    at com[...]BaseProperty.equals([...]:96)
2    at com[...]BaseStringProperty.equals([...]:57)
3    at com[...]BaseCollection.equals([...]:614)
4    at com[...]BaseObject.equals([...]:235)
5    at com[...]XWikiDocument.equalsData([...]:4195)
6    [...]
```

Previous studies [3, 13] show that such kind of test cases are helpful for the developers to debug the application.

For Java programs, the information reported from the operations environment ideally includes a *stack trace*. For instance, Listing 1 presents a stack trace coming from the crash XWIKI-13372.[‡] The stack trace indicates the *exception* thrown (`NullPointerException` here) and the *frames*, *i.e.,* the stack of method calls at the time of the crash, indexed from 1 (at line 1) to 26 (not shown here).

Various approaches use a stack trace as input to automatically generate a test case reproducing the crash. CONCRASH [7] focuses on reproducing *concurrency* failures that violate thread-safety of a class by iteratively generating test code and looking for a thread interleaving that triggers a concurrency crash. JCHARMING [4, 5] applies model checking and program slicing to generate crash reproducing tests. MUCRASH [6] exploits existing test cases written by developers. MUCRASH selects test cases covering classes involved in the stack trace and mutate them to reproduce the crash. STAR [3] applies optimized backward symbolic execution to identify preconditions of a target crash and uses this information to generate a crash reproducing test that satisfies the computed preconditions. Finally, RECORE [14] applies a search-based approach to reproduce a crash using both a stack trace and a core dump, produced by the system when the crash happened, to guide the search.

2.1. Search-based crash reproduction

Search-based approaches have been widely used to solve complex, non-linear software engineering problems, which have multiple and sometimes conflicting optimization objectives [15]. Recently, Soltani *et al.* [8] proposed a search-based approach for crash reproduction called EvoCrash. EvoCrash is based on the EvoSuite approach [16, 17] and applies a new *guided genetic algorithm* to generate a test case that reproduces a given crash using a distance metric, similar to the one described by Rossler *et al.* [14], to guide the search. For a given stack trace, the user specifies a *target frame* relevant to his debugging activities: *i.e.,* the line with a class belonging to his system, from which the stack trace will be reproduced. For instance, applying EvoCrash to the stack trace from Listing 1 with a target frame 2 will produce a crash-reproducing test case for the class `BaseStringProperty` that produces a stack trace with the same two first frames.

An overview of the approach is shown at the right part of Figure 2 (box 5). The first step of this algorithm, called *guided initialization*, is to generate a random population. This random population

---

[‡]Described in issue https://jira.xwiki.org/browse/XWIKI-13372.

is a set of random unit tests where a *target method* call (*i.e.,* the method in the target frame) is injected in each test. During the search, classical guided crossover and guided mutation are applied to the tests in such a way that they ensure that only the tests with a call to the target method are kept in the evolutionary loop. The overall process is guided by a *weighted sum fitness function* [9], applied to each test t:

$$fitness(t) = 3 \times d_l(t) + 2 \times d_e(t) + d_s(t) \tag{1}$$

The terms correspond to the following conditions when executing the test: (i) whether the execution distance from the target line (d_l) is equal to 0.0, in which case, (ii) if the target exception type is thrown (d_e), in which case, (iii) if all frames, from the beginning up until the selected frame, are included in the generated trace (d_s). The overall fitness value for a given test case ranges from 0.0 (crash is fully reproduced) to 6.0 (no test was generated), depending on the conditions it satisfies.

2.2. Seeding strategies for search-based testing

In addition of guided search, a promising technique is *seeding*. Seeding strategies use related knowledge to help the generation process and optimize the fitness of the population [18, 19, 20]. We focus here on the usage of the source code and the available tests as primary sources of information for search-based testing. Other approaches, for instance, search for string inputs on the internet [21], or use the existing test corpus [22] to mine relevant formatted string values (*e.g.,* XML or SQL statements).

2.2.1. Seeding from the source code Three main seeding strategies are exploiting the source code for search-based testing [18, 10, 23]: (i) *constant seeding* uses static analysis to collect and reuse constant values appearing in the source code (*e.g.,* constant values appearing in boundary conditions); (ii) *dynamic seeding* complements constant seeding by using dynamic analysis to collect numerical and string values, observed only during the execution of the software, and reuse them for seeding; and (iii) *type seeding* is used to determine the object type that should be used as an input argument, based on a static analysis of the source code (*e.g.,* by looking at `instanceof` conditions or generic types for instance).

2.2.2. Seeding from the existing tests Rojas *et al.* [10] suggest two test seeding strategies, using *dynamic analysis* on existing test cases: *cloning* and *carving*. Dynamic analysis uses code instrumentation to trace the different methods called during an execution, which, compared to static analysis, makes it easier to identify inter-procedural sequences of method calls (for instance, in the context of a class hierarchy). Cloning and carving have been implemented in EvoSuite and can be used for unit test generation.

For cloning, the execution of an existing test case is copied and used as a member of the initial population of a search process. Specifically, after its instrumentation and execution, the test case is reconstructed internally (without the assertions), based on the execution trace of the instrumented test. This internal representation is then used as-is in the initial population. Internal representation of the cloned test cases are stored in a *test pool*.

For carving, an object is reused during the initialization of the population and mutation of the individuals. In this case, only a subset of an execution trace, containing the creation of a new object
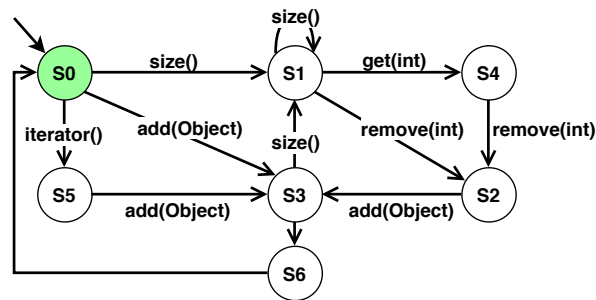
Figure 1. Transition system for method call sequences of the class `java.util.LinkedList` derived from Apache commons math source code and test cases.

and a sequence of methods called on that object, is used to internally build an object on which the methods are called. This object and the subsequent method calls are then inserted as part of a newly created test case (initialization) or in an existing test when a new object is required (mutation). Internal representations of the carved objects[§] are stored in an *object pool*.

The integration of seeding strategies into crash reproduction is illustrated in Figure 2, box 5. As shown, the test cases (respectively objects) to be used by the algorithm are stored in a test case (respectively object) pool, from which they can be used according to user-defined probabilities. For instance, if a test case only contains the creation of a new `LinkedList` (using `new`) that is filled using two `add` method calls, the sequence, corresponding to the execution trace <`new`, `add`, `add`>, may be used as-is in the initial population (cloning) or inserted by a mutation into other test cases (carving).

2.2.3. Challenges in seeding strategies The existing seeding techniques use only one resource to collect information for seeding. However, it is possible that the selected resource does not provide enough information about class usages. For instance, test seeding only uses the carved call sequences from the execution of the existing test cases. If the existing test cases do not cover the behavior of the crash in the interesting classes, this seeding strategy may even misguide the search process. Additionally, if the number of observed call sequences is large, the seeding strategy needs a procedure to prioritize the call sequences for seeding. Using random call sequences as seeds can sometimes misguide the search process. Existing seeding strategies do not currently address these issues.

2.3. Behavioral model-based testing

Model-based testing [11] relies on abstract specifications (models) of the system under test to support the generation of relevant (abstract) test cases. *Transition systems* [24] have been used as a fundamental formalism to reason about test case generation and support the definition of formal test selection criteria [25]. Each abstract test case corresponds to a sequence of method calls on one object: *i.e.,* a path in the transition system starting from the initial state and ending in the initial state, a commonly used convention to deal with finite behaviours [26]. Once selected from the model,

---

[§]In this paper, we use the term *object* to refer to a carved object, *i.e.,* an object plus the sequence of methods called on that object.

abstract test cases are concretized (by mapping the transition system's paths to concrete sequences of method calls) into *executable test cases* to be run on the system. In this paper, we derive abstract test cases (called *abstract object behavior* hereafter) and concretize them, producing pieces of code creating objects and invoking methods on such objects. Those pieces of code serve as seeds for search-based crash reproduction.

Figure 1 shows an example of a transition system representing the possible *sequences* of method calls on `java.util.List` objects. Figure 1 illustrates usages of methods in `java.util.List` objects, learned from the code and tests, in terms of a transition system, from which *sequences* of methods calls can be derived.

The obtained transition system subsumes the behavior of the sequences used to learn it but also allows for new combinations of those sequences. These behaviors are relevant in the context of seeding as the diversity of the objects induced is useful for the search process. Also, generating invalid behaviors from the new combinations is not a problem here as they are detectable during the search process.

2.3.1. Abstract object behavior selection The abstract object behaviors are selected from the transition system according to criteria defined by the tester. In the remainder of this paper, we use *dissimilarity* as selection criteria [27, 28]. Dissimilarity selection, which aims at maximizing the fault detection rate by increasing diversity among test cases, has been shown to be an interesting and scalable alternative to other classical selection criteria [28, 29]. This diversity is measured using a dissimilarity distance (here, 1 - the Jaccard index [30]) between the actions of two abstract object behaviors.

2.3.2. Model Inference The model may be manually specified (and in this case will generally focus on specific aspects of the system) [11], or automatically learned from observations of the system [31, 32, 33, 34, 35, 36]. In the latter case, the model will be incomplete and only contain the *observed behavior* of the system [37]. For instance, the sequence $<$`new`, `addAll` $>$ is valid for a `java.util.List` object but cannot be derived from the transition system in Figure 1 as the `addAll` method call has never been observed. The observed behavior can be obtained via static analysis [38] or dynamically [39]. Model inference may be used for visualization [32, 36], system properties verification [40, 41], or generation [31, 33, 34, 42, 43, 38] and prioritization [26, 44] of test cases.

3. BEHAVIORAL MODEL AND TEST SEEDING FOR CRASH REPRODUCTION

The goal of behavioral model seeding (denoted model seeding hereafter) is to abstract the behavior of the software under test using models and use that abstraction during the search. At the unit test level (which is the considered test generation level in this study), each model is a transition system, like in Figure 1, and represents possible usages of a class: *i.e.,* possible sequences of method calls observed for objects of that class.

The main steps of our model seeding approach, presented in Figure 2, are: the *inference* of the individual models ③ (described in Section 3.1) from the *call sequences* collected through *static*
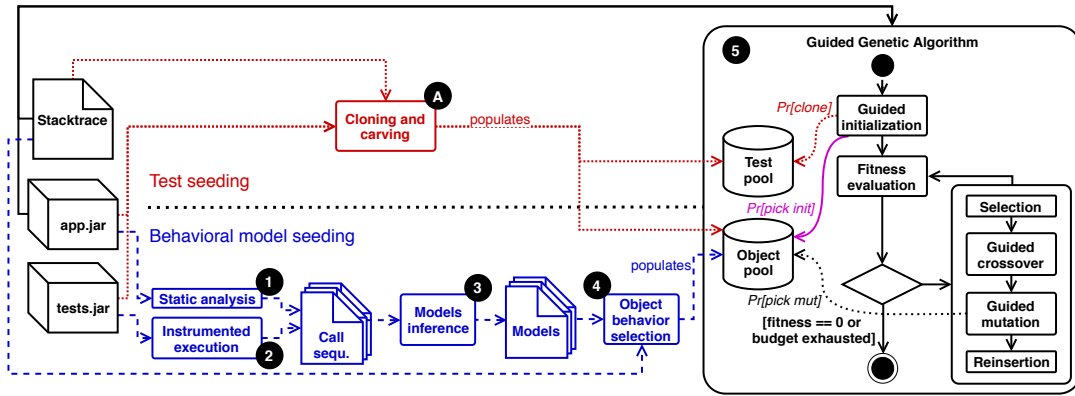
Figure 2. General overview of model seeding and test seeding for search-based crash reproduction

analysis ① performed on the application code (described in Section 3.1.1), and *dynamic analysis* ② of the test cases (described in Section 3.1.2); and for each model, the *selection of abstract object behaviors* ④, that are concretized into Java objects (described in Section 3.2), stored in an *object pool* from which the guided genetic algorithm ⑤ (described in Section 3.3) can randomly pick objects to build test cases during the search process.

3.1. Model inference

Call sequences are obtained by using static analysis on the bytecode of the application ① and by instrumenting and executing the existing test cases ②.

We use n-gram inference to build the transition systems used for model seeding. N-gram inference takes a set of sequences of actions as input to produce a transition system where the nth action depends on the $n-1$ previously executed actions.

A large value of n for the n-gram inference would result in wider transition systems with more states and less incoming transitions, representing a more constrained behavior and producing less diverse test cases. In contrast, a small value of n enables better diversity in the behavior allowed by the model (ending up in more diverse abstract object behaviors), requires less observations to reach stability of the model, simplifies the inference, and results in a more compact model [33, 34]. For these reasons, we use 2-gram inference to build our models.

For each class, the model ③ is obtained using a 2-gram inference method using the call sequences of that class.

For instance, in the transition system of Figure 1, the action `size()`, executed from state s_3 at step k only depends on the fact that the action `add(Object)` has been executed at step $k-1$, independently of the fact that there is a step $k-2$ during which the action `iterator()` has been executed.

Calls to constructors are considered as method calls during model inference. However, constructors may not appear in any transition of the model if no constructor call was observed during the collection of the call sequences. This is usually the case when the call sequences used to infer the model have been captured from objects that are parameters or attributes of a class. If an abstract object behavior does not start by a call to a constructor, a constructor is randomly chosen to initialize the object during the concretization.

For one version of the software under test, the model inference is a one time task. Models can then be directly reused for various crash reproductions.

3.1.1. Static analysis of the application The static analysis is performed on the bytecode of the application. We apply this analysis to all of the available classes in the software under test. In each method of these classes, we build the control flow graph, and for each object of that method, we collect the sequences of method calls on that object. For each object, each path in the control flow graph will correspond to one sequence of method calls. For instance, if the code contains an `if-then-else` statement, the `true` and `false` branches will produce two call sequences. In the case of a loop statement, the `true` branch is considered only once. The static analysis is *intraprocedural*, meaning that only the calls in the current method are considered. If an object is passed as a parameter of a call to a method that (internally) calls other methods on that object, those internal calls will not appear in the call sequences. This analysis ensures collecting all of the existing relevant call sequences for any internal or external class, which is used in the project.

3.1.2. Dynamic analysis for the test cases Since the existing manually developed test cases exemplify potential usage scenarios of the software under test, we apply dynamic analysis to collect all of the transpired sequences during the execution of these scenarios. Contrarily to static analysis, which would require an expensive effort and produce imprecise call sequences, dynamic analysis is *interprocedural*. Meaning that the sequences include calls appearing in the test cases, but also internal calls triggered by the execution of the test case (*e.g.,* if the object is passed as a parameter to a method and methods are internally called on that object). Hence, through dynamic analysis, we gain a more accurate insight into the class usages in these scenarios.

Dynamic analysis of the existing tests is done in a similar way to the carving approach of Rojas *et al.* [10]: instrumentation adds log messages to indicate when a method is called, and the sequences of method calls are collected after execution. In similar fashion to static analysis, we collect call sequences of any observed object (even objects which are not defined in the software under test). The representativeness of the collected sequences depends on the coverage of the existing tests.

3.2. Abstract object behaviors selection

Abstract object behaviors are selected from the transition systems and concretized to populate the object pool used during the search. To limit the number of objects in the pool, we only select abstract object behaviors from two categories of models: models of internal classes (*i.e.,* classes belonging to packages of the software under test) and models of dependency classes (*i.e.,* classes belonging to packages of external dependencies) that are involved in the stack trace. Since we do not seek to validate the implementation of the application, the states are ignored during the selection process.

3.2.1. Selection There exist various criteria to select abstract object behaviors from transition systems [11]. To successfully guide the search, we need to establish a good ratio between *exploration* (the ability to visit new regions of the search space) and *exploitation* (the ability to visit the neighborhood of previously visited regions) [45]. The guided genetic operators which are introduced in EvoCrash approach [8] guarantee the exploitation by focusing the search based on the methods in the stack trace. However, depending on the stack trace, focusing on particular methods

Listing 2: Concretized abstract object behavior for `LinkedList` based on the transition system model of Figure 1

```
1  int[] t = new int[7];
2  t[3] = -2147483647;
3  EuclideanIntegerPoint ep = new EuclideanIntegerPoint(t);
4  LinkedList<[...]> lst = new LinkedList<>();
5  lst.add(ep);
6  lst.add(ep);
```

may reduce the exploration. Poor exploration decreases the diversity of the generated tests and may trap the search process in local optima.

To improve the exploration ability in the search process, we use *dissimilarity* as the criterion to select the abstract object behaviors. Compared to classical structural coverage criteria that seek to cover as many parts of the transition system as possible, dissimilarity tries to increase diversity among the test cases by maximizing a distance d (*i.e.,* the Jaccard index [30]):

$$d = 1 - \frac{\{call_{1i} \in b_1\} \cap \{call_{2j} \in b_2\}}{\{call_{1i} \in b_1\} \cup \{call_{2j} \in b_2\}}$$

Where $b_1 = < call_{11}, call_{12}, \ldots >$ and $b_2 = < call_{21}, call_{22}, \ldots >$ are two abstract object behaviors.

3.2.2. Concretization Each abstract object behavior has to be concretized to an object and method calls before being added to the objects pool. In other words, for each abstract object behavior, if the constructor invocation is not the first action, one constructor is randomly called; and the methods are called on this object in the order specified by the abstract object behavior with randomly generated parameter values. Due to the randomness, each concretization may be different from the previous one. For each abstract object behavior, n concretizations (default value is $n = 1$ to balance scalability and diversity of the objects in the object pool) are done for each abstract object behavior and saved in the object pool. For instance, Listing 2 shows the concretized abstract object behavior `<add(Object), add(Object)>` derived from the transition system model of Figure 1. The type of the parameters (`EuclideanIntegerPoint`) is randomly selected during the concretization and created with required parameter values (an integer array here).

3.3. Guided Initialization and Guided Mutation

Classes are instantiated to create objects during two main steps of the guided genetic algorithm: guided initialization, where objects are needed to create the initial set of test cases; and guided mutation, where objects may be required as parameters when adding a method call. When no seeding is used, those objects are randomly created (as in the concretization step described in Section 3.2.2) by calling the constructor and random methods.

Finally, to preserve exploration in model seeding, objects are picked from the object pool during guided initialization (resp. guided mutation) according to a user-defined probability $Pr[pick\ init]$ (resp. $Pr[pick\ mut]$), and randomly generated otherwise. In our evaluation, we considered four different values for $Pr[pick\ init] \in \{0.2, 0.5, 0.8, 1.0\}$, to study the effect of model seeding on the initialization of the search process. Furthermore, we fixed the value of $Pr[pick\ mut] = 0.3$,

Listing 3: Stack trace excerpt for MATH-79b

```
1    java.lang.NullPointerException
2     at ...KMeansPlusPlusClusterer.assignPointsToClusters()
3     at ...KMeansPlusPlusClusterer.cluster()
```

Listing 4: Test generated for frame 2 of MATH-79b (Listing 3)

```
1   public void testCluster() throws Exception{
2    int[] t = new int[7];
3    t[3] = (-2147483647);
4    EuclideanIntegerPoint ep = new EuclideanIntegerPoint(t);
5    LinkedList<[...]> lst = new LinkedList<>();
6    lst.add(ep);
7    lst.add(ep);
8    KMeansPlusPlusClusterer<[...]> kmean = new KMeansPlusPlusClusterer<>(12);
9    lst.offerFirst(ep);
10   kmean.cluster(lst, 1, (-1357));}
```

corresponding to the default value of $Pr[pick\ mut]$ for test seeding for classical unit test generation in EvoSuite.

As an example of object picking in action, test case generation with model seeding generated the test case in Listing 4 for the second frame of the stack trace from the crash MATH-79b from the Apache commons math project, reported in Listing 3. The target method is the last method called in the test (line 10) and throws a `NullPointerException`, reproducing the input stack trace. The first parameter of the method has to be a `Collection<T>` object. In this case, the guided genetic algorithm picked the list object from the object pool (from Listing 2) and inserted it in the test case (lines 2 to 7). The algorithm also modified that object (during guided mutation) by invoking an additional method on the object (line 9).

3.4. Test seeding

As described in Section 2.2.2, test seeding starts by executing the test cases (Figure 2 box Ⓐ) for carving and cloning, and subsequently populating the test and object pools. Like for model seeding, only internal classes and external classes appearing in the stack trace are considered.

For crash reproduction, the test pool is used only during guided initialization to clone test cases that contain the target class, according to a user-defined $Pr[clone]$ probability. If the target method is not called in the cloned test case, the guided initialization also mutates the test case to add a call to the target method. The object pool is used during the guided initialization and guided mutation to pick objects. As described by Rojas *et al.* [10], the properties of using the object pool during initialization ($Pr[pick\ init]$) and mutation ($Pr[pick\ mut]$) are indicated as a single property called `p_object_pool` in test seeding.

4. IMPLEMENTATION

Relying on the EvoCrash experience [8, 13, 46], we developed Botsing, a framework for crash reproduction with extensibility in mind. Botsing also relies on EvoSuite [47] for the code

instrumentation during test generation and execution by using *evosuite-client* as a dependency. Our open-source implementation is available at `https://github.com/STAMP-project/botsing`. The current version of Botsing includes both test seeding and model seeding as features.

4.1. Test seeding

Test seeding relies on the implementation defined by Rojas *et al.* [10] and available in EvoSuite. This implementation requires the user to provide a list of test cases to consider for cloning and carving. In Botsing, we automated this process using the dynamic analysis of the test cases to automatically detect those accessing classes involved in a given stack trace. We also modified the standard guided initialization and guided mutation to preserve the call to the target method during cloning and carving.

4.2. Model seeding

As mentioned in Section 3, Botsing uses a combination of static and dynamic analysis to infer models. The static analysis (① in Figure 2) uses the reflection mechanisms of EvoSuite to inspect the compiled code of the classes involved in the stack traces, and collect call sequences. The dynamic analysis (② in Figure 2) relies on the test seeding mechanism used for cloning that allows inspecting an internal representation of the test cases obtained after their execution and collect call sequences. The resulting call sequences are then used to infer the transition system models of the classes using a 2-gram inference tool called YAMI [26] (③ in Figure 2). From the infered models, we extract a set of dissimilar (based on the Jaccard distance [30]) abstract object behaviors (④ in Figure 2). For abstract object behavior extraction, we use the VIBeS [48] model-based testing tool. Abstract object behaviors are then concretized into real objects. For this concretization, we rely on the EvoSuite API.

5. EMPIRICAL EVALUATION

Our evaluation aims to assess the effectiveness of each of the mentioned seeding strategies (model and test seeding) on search-based crash reproduction. For this purpose, first, we evaluate the impact of each seeding strategy on the number of reproduced crashes. Second, we examine if using each of these strategies leads to a faster crash reproduction. Third, we see if each seeding strategy can help the search process to start more often. Finally, we characterize the impacting factors of test and model seeding.

Since the focus of this study is using seeding to enhance the guidance of the search initialization, we examine different probabilities of using the seeded information during the guided initialization in the evaluation of each strategy. Hence, we repeat each execution of test seeding with the following values for $Pr[clone]$: 0.2, 0.5, 0.8, and 1.0. Likewise, we execute each execution of model seeding with the same values for $Pr[pick\ init]$ (which is the only property that we can use for modifying the probability of the object seeding in the initialization of model seeding).

5.1. Research questions

In order to assess the usage of test seeding applied to crash reproduction and our new model seeding approach during the guided initialization, we performed an empirical evaluation to answer the two research questions defined in Section 1.

RQ1 *What is the influence of test seeding used during initialization on search-based crash reproduction?* To answer this research question, we compare Botsing executions with *test seeding* enabled to executions where no additional seeding strategy is used (denoted *no seeding* hereafter), from their effectiveness to reproduce crashes and start the search process, the factors influencing this effectiveness, and the impact of test seeding on the efficiency. We divide **RQ1** into four sub-research questions:

RQ1.1 Does test seeding help to reproduce more crashes?

RQ1.2 Does test seeding impact the efficiency of the search process?

RQ1.3 Can test seeding help to initialize the search process?

RQ1.4 Which factors in test seeding impact the search process?

RQ2 *What is the influence of behavioral model seeding used during initialization on search-based crash reproduction?* To answer this question, we compare Botsing executions with *model seeding* to executions with *test seeding* and *no seeding*. We also divide **RQ2** into four sub-research questions:

RQ2.1 Does behavioral model seeding help to reproduce more crashes compared to no seeding?

RQ2.2 Does behavioral model seeding impact the efficiency of the search process compared to no seeding?

RQ2.3 Can behavioral model seeding help to initialize the search process compared to no seeding?

RQ2.4 Which factors in behavioral model seeding impact the search process?

5.2. Setup

5.2.1. Crash selection In a recent study about the evaluation of search-based crash reproduction approaches, Soltani *et al.* [46] introduced a new benchmark, called JCrashPack, containing 200 real-world crashes from seven projects: *JFreeChart*, *Commons-lang*, *Commons-math*, *Mockito*, *Joda-time*, *XWiki*, and *ElasticSearch*. We use the same benchmark for the empirical evaluation of model-seeding and test-seeding on crash reproduction.

To use test and model seeding for reproducing the crashes of JCrashPach, first, we needed to apply static and dynamic analysis on different versions of projects in this benchmark. We successfully managed to run static analysis on all of the classes of JCrashPack. On the contrary, we observed that dynamic analysis was not successful in the execution of existing test suites of ElasticSearch. The reason for this failure stemmed from the technical difficulty of running ElasticSearch tests by the EvoSuite test executor. Since both of the seeding strategies need dynamic analysis, we excluded ElasticSearch cases from JCrashPack for this experiment. JCrashPack contains 124 crashes after excluding ElasticSearch cases. Table I provides more details about our dataset.

We used the selected crashes for the evaluation of *no seeding* and *model seeding*. Since *test seeding* needs existing test cases that are using the target class, we filtered out the crashes which contain only classes without any using tests. Hence, we used only 59 crashes for the evaluation of *test seeding*. More information about average number of used test classes for test seeding is available in Table II.

Table I. Projects used for the evaluation with the number of crashes (**Cr.**), the average number of frames per stack trace ($\overline{\text{frm}}$), the average cyclomatic complexity ($\overline{\text{CCN}}$), the average number of statements ($\overline{\text{NCSS}}$), the average line coverage of the existing test cases ($\overline{\text{LC}}$), and the average branch coverage of the existing test cases ($\overline{\text{BC}}$).

Application	Cr.	$\overline{\text{frm}}$	$\overline{\text{CCN}}$	$\overline{\text{NCSS}}$	$\overline{\text{LC}}$	$\overline{\text{BC}}$
JFreeChart	2	6.00	2.75	63.01k	67%	59%
Commons-lang	22	2.04	3.28	13.38k	91%	87%
Commons-math	27	3.92	2.43	29.98k	90%	84%
Mockito	14	4.64	1.78	6.06k	97%	93%
Joda-Time	8	3.87	2.11	19.41k	89%	82%
XWiki	52	27.25	1.92	177.84k	73%	71%

Table II. Information about test classes and models used, respectively, for test and model seeding in each project. $\overline{test}$ designate the average number of test classes used for test seeding. Also, $\overline{state}$, $\overline{trans}$, and $\overline{BFS}$ denote the average number of states, transitions, and BFS height of the used models, respectively. The standard deviations of each of these metrics (σ) are located beside them.

Project	$\overline{test}$	σ	Project	$\overline{state}$	σ	$\overline{trans}$	σ	$\overline{BFS}$	σ
chart	29.17	20.01	chart	56.67	50.40	157.50	167.86	21.00	17.50
lang	1.45	2.03	lang	39.69	51.49	117.96	158.07	5.58	7.32
math	1.24	1.37	math	14.00	12.46	34.22	40.59	5.20	4.11
mockito	0.73	2.15	mockito	12.18	10.93	21.45	22.70	5.32	3.90
time	9.24	9.55	time	63.35	40.85	230.80	167.99	16.10	11.79
xwiki	0.14	1.09	xwiki	47.94	90.94	139.15	323.75	11.08	17.04

5.2.2. Model inference Since the selected crashes for this evaluation are identified before the model inference process, we have applied the dynamic analysis only on the test cases which use the involved classes in the crashes. During the static analysis, we spot all relevant test cases which call the methods of the classes that have appeared in the stack traces of the crashes. Next, we apply dynamic analysis only on the detected relevant test cases. This filtering process helps us to shorten the model inference execution time without losing accuracy in the generated models.

More information about the inferred models is available in Table II.

5.2.3. Configuration parameters We used a budget of 62,328 fitness evaluations (corresponding on average to 15 minutes of executing Botsing with no seeding on our infrastructure which is introduced in section 5.2.4) to avoid side effects on execution time when executing Botsing on different frames in parallel. We also fixed the population size to 100 individuals as suggested by the latest study on search-based crash reproduction [9]. All other configuration parameters are set at their default value [10], and we used the default weighted sum scalarization fitness function (Equation 1) from Soltani *et al.* [9].

For test seeding executions, as we described at the beginning of this section, we execute each execution with four values for $Pr[clone]$: 0.2 (which is the default value), 0.5, 0.8, and 1.0. Also, we used the default value of 0.3 for p_object_pool.

We also use values 0.2, 0.5, 0.8, and 1.0 for $Pr[pick\ init]$ for model seeding executions. The value of $Pr[pick\ mut]$, which indicates the probability of using seeded information during the mutation, is fixed at 0.3. In addition to model seeding configurations, we fix the size of the selected abstract object behaviors to the size of the individual population in order to ensure that there are enough test cases to initiate the search.

For each frame (951 in total), we executed Botsing for *no seeding* (i.e., no additional seeding compared to the default parameters of Botsing) and each configuration of *model seeding*. Since *test seeding* needs existing test cases which are using the target class, we filtered out the frames that do

not have any test for execution of this seeding strategy. Therefore, we executed each configuration of *test seeding* on the subset of frames (171 in total).

5.2.4. Infrastructure We used 2 clusters (with 20 CPU-cores, 384 GB memory, and 482 GB hard drive) for our evaluation. For each stack trace, we executed an instance of Botsing for each frame which points to a class of the application. We discarded other frames to avoid generating test cases for external dependencies. We ran Botsing on 951 frames from 124 stack traces for no-seeding and each model-seeding strategy configuration. Also, we ran Botsing with test-seeding on 171 frames from 59 crashes. To address the random nature of the evaluated search approaches, we repeated each execution 30 times. We executed a total of 186,560 independent executions for this study. These executions took about 18 days overall.

5.3. Data analysis procedure

To check if the search process can reach a better state using seeding strategies, we analyze the status of the search process after executing each of the cases (each run in one frame of a stack trace). We define 5 states:

  (i) **not started**, the initial population could not be initialized, and the search did not start;
 (ii) **line not reached**, the target line could not be reached;
(iii) **line reached**, the target line has been reached, but the target exception could not be thrown;
(iv) **ex. thrown**, the target line has been reached, and an exception has been thrown but produced a different stack trace; and
 (v) **reproduced** the stack trace could be reproduced.

Since we repeat each execution 30 times, we use the majority of outcomes for a frame reproduction result. For instance, if Botsing reproduces a frame in the majority of the 30 runs, we count that frame as a *reproduced*.

To measure the impact of each strategy in the crash reproduction ratio (**RQ1.1** and **RQ2.1**), we use the Odds Ratio (OR) because of the binary distribution of the related data: a search process either reproduces a crash (the generated test replicates the stack trace from the highest frame which is reproduced by at least one of the other searches) or not. Also, we apply Fisher's exact test, with $\alpha = 0.05$ for the Type I error, to evaluate the significance of results.

Moreover, to answer **RQ1.2** and **RQ2.2**, which investigate the efficiency of the different strategies, we compare the number of fitness function evaluations needed by the search to reach crash reproduction. This metric indicates if seeding strategies lead to better initial populations that need fewer iterations to achieve the crash reproducing test. Since efficiency is only relevant for the reproduced cases, we only applied this comparison on the crashes which are reproduced at least once by no seeding or the seeding strategy (test seeding for RQ1.2 and model seeding for RQ2.2). We use the Vargha-Delaney statistic [49] to appraise the effect size between strategies. In this statistic, a value lower than 0.5 for a pair of factors (A, B) gives that A reduces the number of needed fitness function evaluations, and a value higher than 0.5 shows the opposite. Also, we use the Vargha Delaney magnitude measure to partition the results into three categories having large, medium, and small impact. In addition, to examine the significance of the calculated effect sizes, we use the non-parametric Wilcoxon Rank Sum test, with $\alpha = 0.05$ for Type I error. Moreover, we

Table III. Odds ratios of model/test seeding configurations vs. no seeding in crash reproduction ratio. This table only shows the crashes, which reveal statistically significant differences (p-value < 0.05). An Odds ratio value higher than 1.0 gives that the seeding strategy is better than no seeding, and a value lower than 1.0 shows the opposite.

Conf.	Crash	Odds Ratio (p-value)
test s. 0.5	LANG-6b	Inf (2.37e-02)
	MATH-1b	0.00 (1.69e-17)
	MATH-61b	0.00 (1.69e-17)
	CHART-4b	0.00 (1.69e-17)
	TIME-20b	0.00 (1.94e-03)
	TIME-10b	207.79 (2.36e-12)
	TIME-5b	3.52 (3.52e-02)
test s. 0.8	LANG-6b	Inf (1.94e-03)
	MATH-1b	0.00 (1.69e-17)
	MATH-61b	0.00 (1.69e-17)
	CHART-4b	0.00 (1.69e-17)
	TIME-20b	0.00 (4.64e-05)
	TIME-10b	Inf (9.23e-14)
	TIME-7b	0.00 (6.19e-07)
test s. 1.0	LANG-51b	0.21 (8.21e-03)
	LANG-6b	Inf (4.64e-05)
	MATH-1b	0.00 (1.69e-17)
	MATH-61b	0.00 (1.69e-17)
	CHART-4b	0.00 (1.69e-17)
	TIME-20b	0.00 (5.83e-06)
	TIME-10b	69.79 (2.82e-10)
test s. 0.2	MATH-1b	0.00 (1.69e-17)
	MATH-61b	0.00 (1.69e-17)
	CHART-4b	0.00 (1.69e-17)
	TIME-20b	0.00 (3.19e-04)
	TIME-10b	Inf (9.23e-14)
	TIME-7b	0.00 (1.05e-02)

Conf.	Crash	Odds Ratio (p-value)
model s. 0.2	LANG-9b	Inf (1.94e-03)
	LANG-51b	0.17 (3.33e-03)
	MOCKITO-10b	Inf (1.43e-08)
	XWIKI-13141	13.95 (5.58e-03)
model s. 0.5	LANG-9b	Inf (2.37e-02)
	MOCKITO-10b	Inf (1.87e-07)
	XWIKI-13141	Inf (7.97e-04)
	XWIKI-14152	6.66 (7.41e-03)
model s. 0.8	LANG-9b	Inf (1.94e-03)
	LANG-51b	0.29 (3.70e-02)
	MOCKITO-10b	Inf (8.27e-10)
	XWIKI-13141	Inf (7.97e-04)
	XWIKI-14152	11.24 (2.51e-04)
model s. 1.0	LANG-9b	Inf (1.94e-03)
	MOCKITO-10b	Inf (5.34e-08)
	XWIKI-13141	13.95 (5.58e-03)
	XWIKI-14152	32.80 (5.62e-08)

do note that since the reproduction ratio of each strategy is not 30/30 for each crash, executions that could not reproduce the frame simply reached the maximum allowed budget (62,328).

To measure the impact of each strategy in initializing the first population (**RQ1.3** and **RQ2.3**), we use the same procedure as **RQ1.1** and **RQ2.1** because the distribution of related data in this aspect is binary too (i.e., whether the search process can start the search or not).

For all of the statistical tests in this study, we only use a level of significance $\alpha = 0.05$.

Since the model inference (in model seeding) and test carving (in test seeding) techniques can be applied as one time processes before running any search-based crash reproduction, we do not include them in the efficiency evaluation.

To answer **RQ1.4** and **RQ2.4**, we performed a manual analysis on the logs and crash reproducing test case (if any). We focused our manual analysis on the crash reproduction executions for which the search in one seeding configuration has a significant impact (according to the results of the previous sub-research questions) on (i) *initializing the initial population*, (ii) *crash reproduction*, (iii) or *search process efficiency* compared to no-seeding. Based on our manual analysis, we used a card sorting strategy by assigning keywords to each frame result and grouping those keywords to identify influencing factors.

6. EVALUATION RESULTS

We present the results of the evaluation and answer the two research questions by comparing each seeding strategy with no-seeding.

Table IV. Evaluation results for comparing seeding strategies (test and model seeding) and no-seeding in crash reproduction. $\overline{ratio}$ and σ designate average crash reproduction ratio and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Reproduction		Comparison to no s.		Conf.	Reproduction		Comparison to no s.	
	$\overline{ratio}$	σ	better	worse		$\overline{ratio}$	σ	better	worse
test s. 1.0	23.7	11.01	2	5	model s. 1.0	22.0	11.58	4	0
test s. 0.8	23.4	10.74	2	5	model s. 0.8	21.9	11.92	4	1
test s. 0.5	23.8	10.76	1	4	model s. 0.5	21.8	11.86	4	0
test s. 0.2	23.5	10.93	1	4	model s. 0.2	21.6	12.00	3	1
no s.	25.4	9.65	-	-	no s.	21.3	12.32	-	-

6.1. Test seeding (RQ1)

*6.1.1. Crash reproduction effectiveness (**RQ1.1**)* Figure 3 demonstrates the comparison of each seeding strategy (left-side of the figure is for test seeding and right-side is for model seeding) with the baseline (no seeding). Figures 3a and 3b show the overall comparison, while Figures 3c and 3d illustrate the per project comparison. In each of these figures, the yellow bar shows the number of reproduced crashes in the majority of the 30 executions, and the orange bar shows the non-reproduced crashes.

According to Figure 3a, *test s. 0.8* reproduced the same number of crashes. However, the other configurations of test-seeding reproduced fewer crashes in the majority of times. Moreover, according to Figure 3c, test seeding reproduces one more crash compared to no seeding. Also, some configurations of test seeding can reproduce one extra crash in XWiki and commons-lang projects. On the contrary, all of the configurations of test seeding missed one and two crashes in JFreeChart and commons-math, respectively. Finally, we cannot see any difference between test seeding and no seeding in the Joda-Time project.

Table IV demonstrates the impact of test-seeding on the crash reproduction ratio compared to no-seeding. It indicates that *test s. 0.2 & 0.5* have a better crash reproduction ratio for one of the crashes, while they perform significantly worse in 4 other crashes compared to no-seeding. The situation is almost the same for the other configurations of test seeding: *test s. 0.8 & 1.0* are significantly better in 2 crashes compared to no-seeding. However, they are significantly worse than no-seeding in 5 other crashes. The other interesting point in this table is the standard deviation crash reproduction ratio. This value is slightly higher for all of the test seeding configurations compared to no seeding. The values of odds ratios and and p-values for crashes with significant difference is available in Table III.

The underlying reasons for the observed results in this section are analyzed in RQ1.4.

*6.1.2. Crash reproduction efficiency (**RQ1.2**)* Table V demonstrates the comparison of test-seeding and no-seeding in the number of needed fitness function evaluations for crash reproduction. The average number of fitness function evaluations increases when using test-seeding. It means that test-seeding is slower than no-seeding on average. *test s. 0.8* has the highest average fitness function evaluations.

Moreover, the standard deviations of both no seeding and test seeding are high values (more than 20k evaluations). This notable variation is explainable due to the nature of search-based approaches. In some executions, the initialized population is closer to the objectives, and the search process can achieve reproduction faster. Similar variations are reported in the JCrashPack empirical evaluation
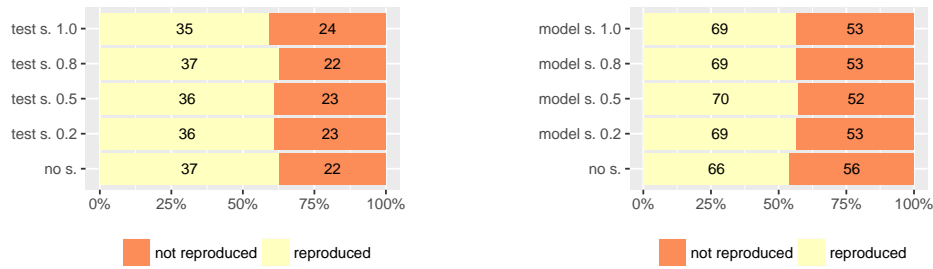
(a) test-seeding vs. no-seeding (for all projects together)

(b) model-seeding vs. no-seeding (for all projects together)
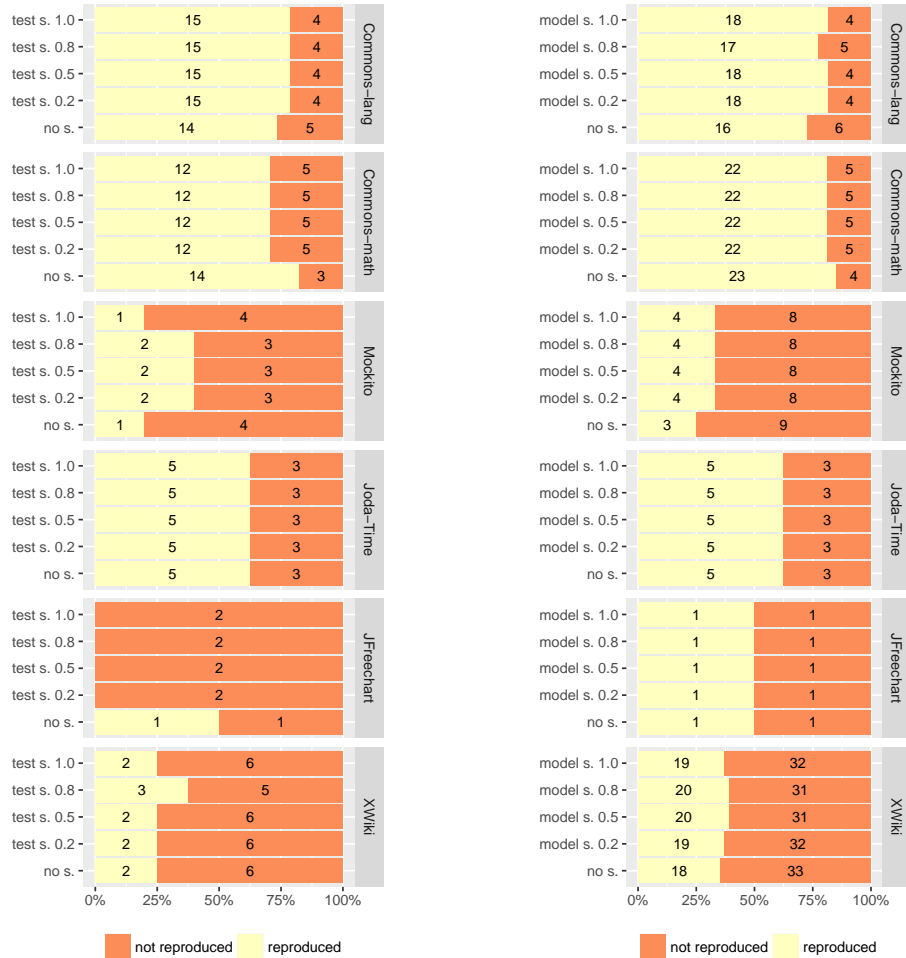
(c) test-seeding vs. no-seeding (per project)

(d) model-seeding vs. no-seeding (per project)

Figure 3. Outcomes observed in the majority of the executions for each crash in total and for each application.

as well [46]. According to the reported standard deviations, we can see that this value increases for all of the configurations of test seeding compared to no seeding.

Also, the values of the effect sizes indicate that the number of crashes that receive (large or medium) positive impacts from *test s. 0.2 & 0.5* for their reproduction speed is higher than the number of crashes that exhibit a negative (large or medium) influence. However, this is not the case

Table V. Evaluation results for comparing test-seeding and no-seeding in the number of fitness evaluations $\overline{\text{evaluations}}$ and σ designate average fitness function evaluations needed for crash reproduction and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

| Conf. | Fitness | | Comparison to no s. | | | | | |
| | | | large | | medium | | small | |
	evaluations	σ	< 0.5	> 0.5	< 0.5	> 0.5	< 0.5	> 0.5
no s.	10,467	22,368.13	-	-	-	-	-	-
test s. 0.2	14,089	25,464	4	3	1	1	2	-
test s. 0.5	13,366	25,043	5	3	1	-	2	1
test s. 0.8	14,254	25,496	3	4	1	5	1	3
test s. 1.0	13,856	25,097	3	13	4	3	1	3

Table VI. Evaluation results for comparing seeding strategies (test and model seeding) and no-seeding in search initialization. $\overline{\text{ratio}}$ and σ designate average successful search initialization ratio and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

| Conf. | Search started | | Comparison to no s. | | Conf. | Search started | | Comparison to no s. | |
	ratio	σ	better	worse		ratio	σ	better	worse
test s. 1.0	26.9	9.22	0	5	model s. 1.0	30.0	0.28	3	0
test s. 0.8	27.9	7.67	1	4	model s. 0.8	30.0	0.00	3	0
test s. 0.5	26.9	9.22	0	5	model s. 0.5	29.7	2.75	2	0
test s. 0.2	27.4	8.49	0	4	model s. 0.2	29.5	3.87	2	1
no s.	29.5	3.94	-	-	no s.	29.2	4.72	-	-

for the other two configurations. In the worst case, *test s. 1.0* is considerably slower than no-seeding (with large effect size) in 13 crashes.

6.1.3. Guided initialization effectiveness (RQ1.3) Table VI indicates the number of crashes where test-seeding had a significant (p-value < 0.05) impact on the search initialization compared to no-seeding. As we can see in this table, any configuration of test-seeding has a negative impact on the search starting process for 4 or 5 crashes. Additionally, this strategy does not have any significant beneficial impact on this aspect except on one crash in *test s. 0.8*. Also, the standard deviation of the average search initialization ratios, in all of the configurations of test seeding, is increased compared to no seeding. For instance, this value for *test s. 0.8* is about three times more than no seeding.

6.1.4. Influencing factors (RQ1.4) To finding the influencing factors in test seeding, we manually analyzed the cases which cause significant differences, in various aspects, between no-seeding and test-seeding. From our manual analysis, we identified 3 factors of the test seeding process that influence the search: (i) **Crash-Test Proximity**, (ii) **Crash-Object Proximity**, and (iii) **Test Execution Cost**.

Crash-Test Proximity For the first factor, we observe that *cloning existing test cases* in the initial population leads to *reproduce new crashes* when the cloned tests include elements which are close to the crash reproducing test. For instance, all of the configurations of test seeding are capable of reproducing the crash LANG 6b, while no-seeding cannot reproduce it. For reproducing this crash, Botsing needs to generate a string of a specific format, and this format is available in the existing test cases, which are seeded to the search process.

However, manually developed tests are not always helpful for crash reproduction. According to the results of Table V, *test s. 1.0*, which always clones test cases, is considerably and largely slower than no-seeding in 13 crashes. In these cases, cloning all of the test cases to form the initial

population can misguide the search process to reach the crash reproducing test. As an example, Botsing needs to generate a simple test case, which calls the target method with an empty string and null object, to reproduce crash LANG-12b. But, *test s. 1.0* clones tests which use the software under test in different ways. To summarize, the overall quality of results of our test seeding solution is highly dependent on the quality of the existing test cases in terms of factors like the distance of existing test cases to the scenario(s) in which the crash occurs and the variety of input data.

Crash-Object Proximity   For the second factor, we observe that (despite the fixed value of $Pr[pick\ mut]$ for test seeding), the objects with call sequences carved from the existing tests and stored in the object pool can help during the search depending on their diversity and their distance from the call sequences that we need for reproducing the given crash. For instance, for crash MATH-4b, Botsing needs to initialize a `List` object with at least two elements before calling the target method in order to reproduce the crash. In test-seeding, such an object had been carved from the existing tests and allowed test seeding to reproduce the crash faster. Also, test-seeding can replicate this crash more frequently: the number of successfully replicated executions, in 30 runs, is higher with test-seeding.

In contrast, the carved objects can misguide the search process for some crashes which need another kind of call sequence. For instance, in crash MOCKITO-9b, Botsing cannot inject the target method into the generated test because the carved objects do not have the proper state to instantiate the input parameters of the target method.

In summary, if the involved classes in a given crash are well-tested (the existing tests contain all of the usage scenarios of these classes), we have more chances to reproduce by utilizing test-seeding.

Test Execution Cost   The third factor points to the challenge of executing the existing test cases for seeding. The related tests for some crashes are either expensive (time/resource consuming) or challenging (due to the security issues) to execute. Hence, the EvoSuite test executor, which is used by Botsing, cannot carve all of them.

As an example of expensive execution, the EvoSuite test executor spends more than 1 hour during the execution of the related test cases for replicating frame 2 of crash Math-1b.

Also, as an example for security issues, the EvoSuite test executor is not successful in running some of the existing tests. It throws an exception during this task. For instance, this executor throws `java.lang.SecurityException` during the execution of the existing test cases for CHART-4b, and it cannot carve any object for seeding.

In some cases, test-seeding faces the mentioned problems during the execution of all of the existing test cases for a crash. If test seeding cannot carve any object from existing tests, there will be no useful call sequence in the object pool to seed during the search process. Hence, although the project contains some potentially valuable test scenarios for reproducing the given crash, there is no difference between no seeding and test seeding in these cases.

*6.1.5. Summary (**RQ1**)*   Test seeding (for any configuration) loses against no-seeding in the search initialization because some of the related test cases of crashes are expensive or even impossible to execute. Also, we observe in the manual analysis that the lack of generality in the existing test cases prevents the crash reproduction search process initialization. In these cases, the carved objects from

the existing tests mismatch the search process in the target method injection. Moreover, this seeding strategy can outperform no seeding in the crash reproduction and search efficiency for some cases (*e.g.,* LANG 6b), thanks to the call sequences carved from the existing tests. However, these carved call sequences can be detrimental to the search process in some cases, if the carved call sequences do not contain beneficial knowledge about crash reproduction, overusing them can misguide the search process.

6.2. Behavioral model seeding (RQ2)

*6.2.1. Crash reproduction effectiveness (**RQ2.1**)* Figure 3b draws a comparison between model-seeding and no-seeding in the crash reproduction ratio according to the results of the evaluation on all of the 124 crashes. As mentioned in Section 5.2.1, since model seeding collects call sequences both from source code and existing tests, it can be applied to all of the crashes (even the crashes that do not have any helpful test). As depicted in this Figure, all of the configurations of model-seeding reproduce more crashes compared to no-seeding in the majority of runs. We observe that *model s. 0.2 & 0.5 & 1.0* reproduce 3 more crashes than no-seeding. In addition, in the best performance of model-seeding, *model s. 0.8* reproduces 70 out of 124 crashes (6% more than no-seeding).

Figure 3d categorizes the results of Figure 3b per application. As we can see in this figure, model seeding replicates more crashes for XWiki, commons-lang, and Mockito. However, no-seeding reproduces one crash more than model-seeding for commons-math. For the other projects, the number of reproduced crashes does not change between no-seeding and different configurations of model-seeding.

We also check how many crashes can be reproduced at least once with model seeding, but not with no seeding. In total, model-seeding configurations reproduce nine new crashes that no-seeding cannot reproduce.

Table IV indicates the impact of model-seeding on the crash reproduction ratio. As we can see in this table, *model s. 0.2* has a significantly better crash reproduction ratio in 3 crashes. Also, other configurations of model-seeding are significantly better than no seeding in 4 crashes. This improvement is achieved by model-seeding, while 2 out of 4 configurations of model-seeding have a significant unfavorable impact on only one crash. The values of odds ratios and and p-values for crashes with significant difference is available in Table III.

*6.2.2. Crash reproduction efficiency (**RQ2.2**)* Table VII compares the number of the needed fitness function evaluations for crash reproduction in model-seeding and no-seeding. As we can see in this table, the average effort is reduced by using model-seeding. On average *mode s. 1.0* achieves the fastest crash reproduction.

According to this table, and in contrast to test-seeding, model-seeding's efficiency is slightly positive. The number of crashes that model-seeding has a positive large or medium influence (as Vargha Delaney measures are lower than 0.5) on varies between 3 to 5. Also, model-seeding has a large adverse effect size (as Vargha Delaney measures are higher than 0.5) on one crash, while this number is higher for test-seeding (e.g., 13 for *test s. 1.0*).

Table VII does not include the cost of model generation for seeding as mentioned in our experimental setup. In our case, model generation was not a burden and is performed only once per case study. We will cover this point in more detail in Section 7.

Table VII. Evaluation results for comparing model-seeding and no-seeding in the number of fitness evaluations evaluations and σ designate average fitness function evaluations needed for crash reproduction and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

| Conf. | Fitness | | Comparison to no s. | | | | | |
| | | | large | | medium | | small | |
	evaluations	σ	< 0.5	> 0.5	< 0.5	> 0.5	< 0.5	> 0.5
no s.	18,713.1	28,023.93	-	-	-	-	-	-
model s. 0.2	18,016.1	27,699.61	2	1	1	1	2	1
model s. 0.5	17,646.9	27,463.02	2	1	2	-	2	1
model s. 0.8	17,564.5	27,400.27	3	1	2	-	1	3
model s. 1.0	17,268.8	27,190.73	3	1	2	-	1	2

*6.2.3. Guided initialization effectiveness (**RQ2.3**)* Table VI provides a comparison between model-seeding and no-seeding in the search initialization ratio. As shown in this Table, *model s. 0.2 & 0.5* significantly outperform no seeding in starting the search process for two crashes. This number increases to 3 for *model s. 0.8 & 1.0*. In contrast to test-seeding, most of the configurations of model-seeding do not have any significant negative impact on the search initialization (only *model s. 0.2* is significantly worse than no-seeding in one crash). Notably, the average search initialization ratios for all of the model seeding configurations are slightly higher than no seeding. In the best case for model seeding, *model s. 0.8 & 1.0* is 30/30 runs, and the standard deviations for these two configurations are 0 or close to 0.

*6.2.4. Influencing factors (**RQ2.4**)* We have manually analyzed the crashes which lead to significant differences between different configurations of model seeding and no seeding. In doing so, we have identified 4 influencing factors in model-seeding on search-based crash reproduction, namely: (i) using **Call sequence dissimilarity** for guided initialization, (ii) having **Information source diversity** to infer the behavioral models, (iii) **Sequence priority** for seeding by focusing on the classes involved in the stack trace, and (iv) having **Fixed size abstract object behavior selection** from usage models.

Call sequence dissimilarity   Using *dissimilar call sequences* to populate the object pool in model seeding seems particularly useful for search efficiency compared to test seeding. In particular, if the number of test cases is large, model seeding enables to (re)capture the behavior of those tests in the model and regenerate a smaller set of call sequences which maximize diversity, augmenting the probability to have more diverse objects used during the initialization. For instance, Botsing with model-seeding is statistically more efficient than other strategies for replicating crash XWIKI-13141. Through our manual analysis we observed that model-seeding could replicate crash XWIKI-13141 in the initial population in 100% of cases, while the other seeding strategies replicate it after a couple of iterations. In this case, despite the large size of the target class behavioral model (35 transitions and 17 states), the diversity of the selected abstract object behaviors guarantees that Botsing seeds the reproducing test cases to the initial population.

Information source diversity   Having *multiple sources* to infer the model from helps to select diversified call sequences compared to test seeding. For instance, the sixth frame of the crash XWIKI-14556 points to a class called `HqlQueryExecutor`. No seeding cannot replicate this

crash because it does not have any guidance from existing solutions. Also, since the test carver could not detect any existing test which is using the related classes, this seeding strategy does not have any knowledge to achieve reproduction. In contrast, the knowledge required for reproducing this crash is available in the source code, and model-seeding learned it from static analysis of this resource. Hence, this seeding strategy is successful in accomplishing crash reproduction.

Sequence priority   By *prioritizing classes* involved in the stack trace for the abstract object behaviors selection, the object pool contains more objects likely to help to reproduce the crash. For instance, for the 10th frame of the crash LANG-9b, model seeding could achieve reproduction in the majority of runs, compared to 0 for test and no seeding, by using the class `FastDateParser` appearing in the stack trace.

Fixed size abstract object behavior selection   The last factor points to the fixed number of the generated abstract object behaviors from each model. In some cases, we observed that model-seeding was not successful in crash reproduction because the usage models of the related classes were large, and it was impossible to cover all of the paths with 100 abstract object behaviors. As such, this seeding strategy missed the useful dissimilar paths in the model. As an example, model-seeding was not successful in replicating crash XWIKI-8281 (which is replicated by no-seeding and test-seeding). In this crash, the unfavorable generated abstract object behaviors for the target class misguided the search process in model seeding.

*6.2.5. Summary (**RQ2**)* Model seeding achieves a better search initialization ratio compared to no seeding. With respect to the best achievement of model seeding (*model s. 0.8 & 1.0*), they decrease the number of not started searches in 3 crashes. Moreover, compared to no seeding, model seeding increases the number of crashes that can be reproduced in the majority of times to 6%. It also reproduces 9 (out of 124) extra crashes that are unreproducible with no-seeding. In addition, model seeding improves the efficiency of search-based crash reproduction compared to no seeding. It takes, on average, less fitness function evaluations. Also, model seeding delivers more positive significant impact on the efficiency of the search process compared to no seeding.

In general, model seeding outperforms no seeding in all of the aspects of search-based crash reproduction. According to the manual analysis that we have performed in this study, model seeding achieves this performance thanks to multiple factors: Call sequence dissimilarity, Information source diversity, and Sequence priority. Nevertheless, we observe a negative impacting factor in model seeding, as well. This factor is the fixed size abstract object behavior selection.

7. DISCUSSION

7.1. Practical implications

Model derivation costs.   Generating seeds comes with a cost. For our worst case, XWIKI-13916, we collected 286K call sequences from static and dynamic analysis and generated 7,880 models from which we selected 6K abstract object behaviors. We repeated this process 10 times and found the average time for call sequence collection to be 14.2 seconds; model inference took 77.8 seconds;

and abstract object behavior selection and concretization took 51.5 seconds. We do note however that the model inference is a one-time process that could be done offline (in a continuous integration environment). After the initial inference of models, any search process can utilize model seeding. To summarize, the total initial overhead is ~ 2.5 minutes, and the total nominal overhead is around ~ 1.25 minute. We argue that **the overhead of model seeding is affordable giving its increased effectiveness.** The initial model inference can also be incremental, to avoid complete regeneration for each update of the code, or limited to subparts of the application (like in our evaluation where we only applied static and dynamic analysis for classes involved in the stack trace). Similarly, abstract object behavior selection and concretization may be prioritized to use only a subset of the classes and their related model. In our current work, this prioritization is based on the content of the stack traces. Other prioritization heuristics, based for instance on the size of the model (reflecting the complexity of the behavior), is part of our future work.

Applicability and effectiveness.   Generally, test seeding alone does not make crash reproduction more effective. Actually, test seeding has a more negative impact on the search-based crash reproduction. Test seeding only uses dynamic analysis, which entails that it collects more accurate information from the potential usage scenarios of the software under test; it also means that this strategy collects more limited information for seeding. If these limited amounts of call sequences differ from the call sequences needed to reproduce the crash scenario, test seeding can misguide the crash reproduction search process.

In contrast to test-seeding, we observe that model seeding always performs better than no seeding with different configurations. As such, we observe that **model seeding can reproduce more crashes than other strategies**. Also, since model seeding also exploits test cases, thereby subsuming test seeding regarding the observed behavior of the application that is reused during the search, greater performance can be attributed to the analysis of the source code translated in the model.

In our experiments, various configurations of model seeding reproduced 8 new crashes that neither test seeding nor no seeding strategies could reproduce. Additionally, **only model seeding could reproduce stack traces with more than seven frames** (*e.g.,* LANG-9b). Still, model seeding missed the reproduction of one crash which is reproduced by no seeding. Despite the achieved improvements by model seeding, this seeding strategy could not outperform no-seeding dramatically (crash reproduction improved by 6%). To better understand the reasons for the results, we manually analyzed the logs of Botsing executions on the crashes for which model seeding could not show any improvements. Through this investigation, we noticed that the generated usage models in these cases are limited and they do not contain the beneficial call sequences for covering the particular path that we need for crash reproduction. The average size of the generated model in this study is 7 states and 14 transitions. We believe that by collecting more call sequences from different sources (*i.e.,* log files), model seeding can increase the number of crash reproductions.

Also, we observe that the size of the generated abstract behaviors set is commensurate to the size of the inferred model. If we have a small model, and we choose too many abstract behaviors, we will get similar abstract behaviours that misguide the search process. In contrast, if we chose a small set of abstract behaviors from a behavioral model with a large size, we will miss the chance of using all of the potential of the model for increasing the chance of crash reproduction by the search process.

Table VIII. Evaluation results for comparing different configurations of model seeding in crash reproduction. $\overline{\text{rate}}$ and σ designate average crash reproduction rate and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Reproduction		Comparison to other conf.	
	rate	σ	better	worse
Pr[init]=0.0 Pr[mut]=0.3	18.8	13.81	0	11
Pr[init]=0.0 Pr[mut]=0.6	19.0	13.64	0	10
Pr[init]=0.0 Pr[mut]=0.9	19.0	13.55	0	13
Pr[init]=0.2 Pr[mut]=0.0	20.4	12.42	2	2
Pr[init]=0.2 Pr[mut]=0.3	19.6	12.87	0	7
Pr[init]=0.2 Pr[mut]=0.6	19.8	12.88	1	5
Pr[init]=0.2 Pr[mut]=0.9	19.4	13.15	0	7
Pr[init]=0.5 Pr[mut]=0.0	20.8	12.17	3	1
Pr[init]=0.5 Pr[mut]=0.3	20.6	12.29	3	2
Pr[init]=0.5 Pr[mut]=0.6	19.4	13.24	0	7
Pr[init]=0.5 Pr[mut]=0.9	20.0	12.58	1	5
Pr[init]=0.8 Pr[mut]=0.0	21.8	11.46	8	0
Pr[init]=0.8 Pr[mut]=0.3	21.6	11.53	6	0
Pr[init]=0.8 Pr[mut]=0.6	21.8	11.77	8	0
Pr[init]=0.8 Pr[mut]=0.9	20.8	11.96	3	2
Pr[init]=1.0 Pr[mut]=0.0	21.6	11.53	6	0
Pr[init]=1.0 Pr[mut]=0.3	23.0	11.31	12	0
Pr[init]=1.0 Pr[mut]=0.6	21.6	11.82	8	0
Pr[init]=1.0 Pr[mut]=0.9	22.6	11.30	11	0

Extendability. The usage models can be inferred from any resource providing call sequences. In this study, we used the call sequences derived from the source code and existing test cases. However, we can extend the models with extra resources (*e.g.,* execution logs). Also, the abstract object behavior selection approach can be adapted according to the problem. In this study, we used the dissimilarity strategy to increase the diversity of the generated tests. Moreover, model seeding makes a distinction between using the object pool during guided initialization and guided mutation (as shown in Figure 2). This distinction enables us to study the influence of seeding during the different steps of the algorithm independently.

7.2. Model seeding configuration

Model seeding can be configured with different $Pr[pick\ init]$ and $Pr[pick\ mut]$ probabilities. Like many other parameters in search-based test case generation [50], the values of those parameters could influence our results. Although a full investigation of the effect of $Pr[pick\ init]$ and $Pr[pick\ mut]$ on the search process is beyond the scope of this paper, we set up a small experiment on a subset of crashes (10 crash in total) with 15 new configurations, each one run 10 times.

Tables VIII and IX presents the configurations used for $Pr[pick\ init]$ and $Pr[pick\ mut]$ with, for each one, the crash reproduction effectiveness (Table VIII), and the crash reproduction efficiency (Table IX). In general, we observe that changing the probability of picking an object during guided initialization ($Pr[pick\ init]$) has an impact on the search and leads to more reproduced crashes with a lower number of fitness evaluations. This confirms the results presented in Section 6. Changing the probability of picking an object during mutation ($Pr[pick\ mut]$) does not seem to have a large impact on the search. A full investigation of the effects of $Pr[pick\ init]$ and $Pr[pick\ mut]$ on the search process is part of our future work.

Table IX. Evaluation results for comparing different configurations of model seeding in the number of fitness evaluations $\overline{\text{rate}}$ and σ designate average fitness function evaluations needed for crash reproduction and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

| Conf. | Fitness | | Comparison to other configurations | | | | | |
| | | | large | | medium | | small | |
	evaluations	σ	< 0.5	> 0.5	< 0.5	> 0.5	< 0.5	> 0.5
Pr[init]=0.0 Pr[mut]=0.3	23,456.5	30,105.20	-	5	-	3	-	3
Pr[init]=0.0 Pr[mut]=0.6	23,066.3	29,976.23	-	2	-	5	-	3
Pr[init]=0.0 Pr[mut]=0.9	23,030.9	30,001.82	-	7	-	4	1	2
Pr[init]=0.2 Pr[mut]=0.0	20,179.0	29,012.80	-	-	1	2	1	-
Pr[init]=0.2 Pr[mut]=0.3	21,803.0	29,620.34	-	2	2	5	-	1
Pr[init]=0.2 Pr[mut]=0.6	21,448.9	29,441.74	-	2	-	3	1	2
Pr[init]=0.2 Pr[mut]=0.9	22,214.6	29,752.12	-	2	2	5	1	1
Pr[init]=0.5 Pr[mut]=0.0	19,371.3	28,668.58	-	-	2	1	3	-
Pr[init]=0.5 Pr[mut]=0.3	19,766.8	28,849.00	-	-	1	2	2	-
Pr[init]=0.5 Pr[mut]=0.6	22,245.2	29,729.80	-	-	-	4	-	3
Pr[init]=0.5 Pr[mut]=0.9	21,030.0	29,302.03	-	2	-	3	1	2
Pr[init]=0.8 Pr[mut]=0.0	17,329.0	27,693.98	2	-	6	-	-	1
Pr[init]=0.8 Pr[mut]=0.3	17,710.5	27,919.28	1	-	4	-	3	-
Pr[init]=0.8 Pr[mut]=0.6	17,327.0	27,694.60	2	-	6	-	-	-
Pr[init]=0.8 Pr[mut]=0.9	19,383.3	28,659.38	-	-	1	1	2	2
Pr[init]=1.0 Pr[mut]=0.0	17,730.5	27,906.92	1	-	4	-	3	1
Pr[init]=1.0 Pr[mut]=0.3	14,863.9	26,275.53	7	-	3	-	-	-
Pr[init]=1.0 Pr[mut]=0.6	17,692.5	27,930.17	2	-	5	-	1	-
Pr[init]=1.0 Pr[mut]=0.9	15,656.9	26,798.15	7	-	5	-	1	-

7.3. Threats to validity

7.3.1. Internal validity We selected 124 crashes from 5 open source projects: 33 crashes have previously been studied [9] and we added additional crashes from Xwiki and Defects4J (see Section 5). Since we focused on the effect of seeding during guided initialization, we fixed the $Pr[pick\ mut]$ value (which, due to the current implementation of Botsing, is also used as $Pr[pick\ init]$ value in test seeding) to 0.3, the default value used in EvoSuite for unit test generation. The effect of this value for crash reproduction, as well as the usage of test and model seeding in guided initialization, is part of our future work. We cannot guarantee that our extension of Botsing is free of defects. We mitigated this threat by testing the extension and manually analyzing a sample of the results. Finally, each frame has been run 30 times for each seeding configuration to take randomness into account and we derive our conclusions based on standard statistical tests [51, 52].

7.3.2. External validity We cannot guarantee that our results are generalizable to all crashes. However, we used JCrashPack, which is the most recent benchmark for Java crash reproduction. This benchmark is assembled carefully from seven Java projects and contains 200 real-life crashes. Since the EvoSuite test executor is unsuccessful in running the existing test cases of one of the seven projects in JCrashPack (ElasticSearch), thereby test seeding and dynamic analysis of model-seeding are not applicable on crashes of this project, we excluded ElasticSearch crashes from JCrashPack. The diversity of crashes in this benchmark also suggests mitigation of this threat.

7.3.3. Verifiability A replication package of our empirical evaluation is available at `https://github.com/STAMP-project/ExRunner-bash/tree/master`. The complete results and analysis scripts are also provided in this package. Our extension of Botsing is released under a LGPL 3.0 license and available at `https://github.com/STAMP-project/botsing`.

8. FUTURE WORK

We observed that one of the advantageous factors in model seeding, which helps the search process to reproduce more crashes, consists in using more multiple resources for collecting the call sequences. Further diversification of sources is worth considering. In our future work, we will consider other sources of information, like logs of the running environment, to collect relevant call sequences and additional information about the actual usage of the application.

Also, collecting additional information from the log files would enable using full-fledged *behavioral usage models* (*i.e.,* a transition system with probabilities on their transitions quantifying the actual usage of the application) to select and *prioritize* abstract object behaviors according to that usage as it is suggested by statistical testing approaches [26]. For instance, we can put a high priority for the most uncommon observed call sequences for the abstract object behavior selection. We observed that selecting the most dissimilar paths in model-seeding helps the search process through crash reproduction. However, there is no guarantee that this approach is the best one. In future studies, we examine this approach with the new abstract object behavior selection approaches that we gain by the new full-fledged *behavioral usage models*.

In this study, we focus on the impact of seeding during guided initialization by using different values for $Pr[pick\ init]$ and $Pr[clone]$ and setting $Pr[pick\ mut]$ to the default value (0.3). However, our results show that even with the default value 0.3, using seeded objects during the search process helps to reproduce several crashes. Our future work includes a thorough assessment of that factor. Furthermore, in the current version of model seeding, we noticed that the fixed size for the selected abstract object behaviors from the usage models could negatively impact the crash reproduction process. This set's size affects Botsing's performance and must be chosen carefully. If too small, abstract object behaviors may not cover the transition system sufficiently, missing out on important usage information. Too few abstract object behaviors can misguide the search process. In contrast, too many of them will lead to a time-consuming test concretization process. In future investigations, we will study the integration of the search process with the abstract object behavior selection from the models. This integration can guide the seeding (*e.g.,* the abstract object behavior selection) using the current status of the search process.

Finally, we hypothesize that this seeding strategy may be useful for other search-based software testing applications and we will evaluate this hypothesis in our future work.

9. CONCLUSION

Manual crash reproduction is labor-intensive for developers. A promising approach to alleviate them from this challenging activity is to automate crash reproduction using search-based techniques. In this paper, we evaluate the relevance of using both test and behavioral model seeding to improve crash reproduction achieved by such techniques. We implement both test seeding and the novel model seeding in Botsing.

For practitioners, the implication is that more crashes can be automatically reproduced, with a small cost. In particular, our results show that behavioral model seeding outperforms test seeding and no seeding without a major impact on efficiency. The different behavioral model seeding

configurations reproduce 6% more crashes compared to no seeding, while test seeding reduces the number of reproduced crashes. Also, behavioral model seeding can significantly increase the search initialization rate for 3 crashes compared to no seeding, while test seeding performs worse than no seeding in this aspect. We hypothesize that the achieved improvements by model seeding can be further extended by using more resources (*i.e.,* execution logs) for collecting the call sequences which are beneficial for the model generation.

From the research perspective, by abstracting behavior through models and taking advantage of the advances made by the model-based testing community, we can enhance search-based crash reproduction. Our analysis reveals that (1) using collected call sequences, together with (2) the dissimilar selection, and (3) prioritization of abstract object behaviors, as well as (4) the combined information from source code and test execution, enable more search processes to get started, and ultimately more crashes to be reproduced.

In our future work, we will explore whether behavioral model seeding has further ranging implications for the broader area of search-based software testing. Furthermore, we aim to study the effect of changing the seeding probabilities on the search process, explore other sources of data to generate the model and try different abstract object behavior selection strategies.

REFERENCES

1. Zeller A. *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition*. Academic Press, 2009.
2. Beller M, Spruit N, Spinellis D, Zaidman A. On the dichotomy of debugging behavior among programmers. *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 2018; 572–583.
3. Chen N, Kim S. STAR: Stack trace based automatic crash reproduction via symbolic execution. *IEEE Transactions on Software Engineering* 2015; **41**(2):198–220, doi:10.1109/TSE.2014.2363469.
4. Nayrolles M, Hamou-Lhadj A, Tahar S, Larsson A. JCHARMING: A bug reproduction approach using crash traces and directed model checking. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, 2015; 101–110, doi:10.1109/SANER.2015.7081820.
5. Nayrolles M, Hamou-Lhadj A, Tahar S, Larsson A. A bug reproduction approach based on directed model checking and crash traces. *Journal of Software: Evolution and Process* mar 2017; **29**(3):e1789, doi:10.1002/smr.1789.
6. Xuan J, Xie X, Monperrus M. Crash reproduction via test case mutation: let existing test cases help. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, ACM Press, 2015; 910–913, doi:10.1145/2786805.2803206.
7. Bianchi FA, Pezzè M, Terragni V. Reproducing concurrency failures from crash stacks. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, ACM Press, 2017; 705–716, doi:10.1145/3106237.3106292.
8. Soltani M, Panichella A, van Deursen A. A Guided Genetic Algorithm for Automated Crash Reproduction. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, 2017; 209–220, doi:10.1109/ICSE.2017.27.
9. Soltani M, Derakhshanfar P, Panichella A, Devroey X, Zaidman A, van Deursen A. Single-objective versus Multi-Objectivized Optimization for Evolutionary Crash Reproduction. *Proceedings of the 10th Symposium on Search-Based Software Engineering (SSBSE)*, Springer, 2018.
10. Rojas JM, Fraser G, Arcuri A. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability* aug 2016; **26**(5):366–401, doi:10.1002/stvr.1601.

11. Utting M, Legeard B. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007.

12. White M, Vásquez ML, Johnson P, Bernal-Cárdenas C, Poshyvanyk D. Generating reproducible and replayable bug reports from android application crashes. *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015*, Lucia AD, Bird C, Oliveto R (eds.), IEEE Computer Society, 2015; 48–59, doi:10.1109/ICPC.2015.14.

13. Soltani M, Panichella A, Van Deursen A. Search-Based Crash Reproduction and Its Impact on Debugging. *IEEE Transactions on Software Engineering* 2018; doi:10.1109/TSE.2018.2877664.

14. Rößler J, Zeller A, Fraser G, Zamfir C, Candea G. Reconstructing core dumps. *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation, ICST 2013*, IEEE, 2013; 114–123, doi:10.1109/ICST.2013.18.

15. Harman M, Mansouri SA, Zhang Y. Search-based software engineering. *ACM Computing Surveys* nov 2012; **45**(1):1–61, doi:10.1145/2379776.2379787.

16. Fraser G, Arcuri A. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* feb 2013; **39**(2):276–291, doi:10.1109/TSE.2012.14.

17. Fraser G, Arcuri A. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology* dec 2014; **24**(2):1–42, doi:10.1145/2685612.

18. Fraser G, Arcuri A. The Seed is Strong: Seeding Strategies in Search-Based Software Testing. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, IEEE, 2012; 121–130, doi:10.1109/ICST.2012.92.

19. Chen T, Li M, Yao X. On the Effects of Seeding Strategies: A Case for Search-based Multi-Objective Service Composition. *Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '18*, ACM Press, 2018; 1419–1426, doi:10.1145/3205455.3205513.

20. Lopez-Herrejon RE, Ferrer J, Chicano F, Egyed A, Alba E. Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of Software Product Lines. *2014 IEEE Congress on Evolutionary Computation (CEC)*, IEEE, 2014; 387–396, doi:10.1109/CEC.2014.6900473.

21. McMinn P, Shahbaz M, Stevenson M. Search-Based Test Input Generation for String Data Types Using the Results of Web Queries. *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12)*, IEEE, 2012; 141–150, doi:10.1109/ICST.2012.94.

22. Toffola LD, Staicu CA, Pradel M. Saying Hi!' is not enough: Mining inputs for effective test generation. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2017; 44–49, doi:10.1109/ASE.2017.8115617.

23. Alshahwan N, Harman M. Automated web application testing using search based software engineering. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, IEEE, 2011; 3–12, doi:10.1109/ASE.2011.6100082.

24. Baier C, Katoen JP. *Principles of Model Checking*. MIT Press, 2007.

25. Tretmans J. Model based testing with labelled transition systems. *Formal methods and testing* 2008; :1–38.

26. Devroey X, Perrouin G, Cordy M, Samih H, Legay A, Schobbens PY, Heymans P. Statistical prioritization for software product line testing: an experience report. *Software & Systems Modeling* feb 2017; **16**(1):153–171, doi:10.1007/s10270-015-0479-8.

27. Cartaxo EG, Machado PDL, Neto FGO. On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification and Reliability* 2011; **21**(2):75–100, doi:10.1002/stvr.413.

28. Hemmati H, Arcuri A, Briand L. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology* feb 2013; **22**(1):1–42, doi:10.1145/2430536.2430540.

29. Mondal D, Hemmati H, Durocher S. Exploring Test Suite Diversification and Code Coverage in Multi-Objective Test Case Selection. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, ICST '15, IEEE, 2015; 1–10, doi:10.1109/ICST.2015.7102588.

30. Jaccard P. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Société Vaudoise des Sciences Naturelles* 1901; **37**:547–579, doi:10.5169/seals-266450.

31. Herbold S, Harms P, Grabowski J. Combining usage-based and model-based testing for service-oriented architectures in the industrial practice. *International Journal on Software Tools for Technology Transfer* jun 2017; **19**(3):309–324, doi:10.1007/s10009-016-0437-y.

32. Leemans M, van der Aalst WMP, van den Brand MGJ. The Statechart Workbench: Enabling scalable software event log analysis using process mining. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2018; 502–506, doi:10.1109/SANER.2018.8330248.

33. Sprenkle SE, Pollock LL, Simko LM. Configuring effective navigation models and abstract test cases for web applications by analysing user behaviour. *Software Testing, Verification and Reliability* 2013; **23**(6):439–464, doi:10.1002/stvr.1496.

34. Sprenkle S, Pollock L, Simko L. A Study of Usage-Based Navigation Models and Generated Abstract Test Cases for Web Applications. *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, IEEE, 2011; 230–239, doi:10.1109/ICST.2011.34.

35. Tonella P, Tiella R, Nguyen CD. Interpolated n-grams for model based testing. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, ACM Press, 2014; 562–572, doi:10.1145/2568225.2568242.

36. Verwer S, Hammerschmidt CA. flexfringe: A Passive Automaton Learning Package. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, O'Conner L (ed.), IEEE, 2017; 638–642, doi:10.1109/ICSME.2017.58.

37. Tonella P, Marchetto A, Nguyen CD, Jia Y, Lakhotia K, Harman M. Finding the optimal balance between over and under approximation of models inferred from execution logs. *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation, ICST 2012*, IEEE, 2012; 21–30, doi:10.1109/ICST.2012.82.

38. Fraser G, Zeller A. Exploiting Common Object Usage in Test Case Generation. *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, IEEE, 2011; 80–89, doi:10.1109/ICST.2011.53.

39. Krka I, Brun Y, Popescu D, Garcia J, Medvidovic N. Using dynamic execution traces and program invariants to enhance behavioral model inference. *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, ACM, 2010; 179–182, doi:10.1145/1810295.1810324.

40. Lorenzoli D, Mariani L, Pezzè M. Automatic generation of software behavioral models. *Proceedings of the 13th international conference on Software engineering - ICSE '08*, ACM Press, 2008; 501, doi:10.1145/1368088.1368157.

41. Ghezzi C, Pezzè M, Sama M, Tamburrelli G. Mining Behavior Models from User-intensive Web Applications. *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, ACM Press: Hyderabad, India, 2014; 277–287, doi:10.1145/2568225.2568234.

42. Prowell S, Poore J. Computing system reliability using Markov chain usage models. *Journal of Systems and Software* oct 2004; **73**(2):219–225, doi:10.1016/S0164-1212(03)00241-3.

43. Zhang Y, Harman M, Jia Y, Sarro F. Inferring Test Models from Kate's Bug Reports Using Multi-objective Search. *Search-Based Software Engineering, SSBSE '15*, Barros M, Labiche Y (eds.), Springer International Publishing, 2015; 301–307.

44. Dulz W, Fenhua Zhen. MaTeLo - statistical usage testing by annotated sequence diagrams, Markov chains and TTCN-3. *Third International Conference on Quality Software, 2003. Proceedings.*, IEEE, 2003; 336–342, doi:10.1109/QSIC.2003.1319119.

45. Črepinšek M, Liu SH, Mernik M. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Computing Surveys (CSUR)* 2013; **45**(3):35.

46. Soltani M, Derakhshanfar P, Devroey X, van Deursen A. A benchmark-based evaluation of search-based crash reproduction. *Empirical Software Engineering* Aug 2019; doi:10.1007/s10664-019-09762-1. URL https://doi.org/10.1007/s10664-019-09762-1.

47. Fraser G, Arcuri A. EvoSuite: automatic test suite generation for object-oriented software. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, ACM Press, 2011; 416–419, doi:10.1145/2025113.2025179.

48. Devroey X, Perrouin G, Legay A, Schobbens PY, Heymans P. Search-based Similarity-driven Behavioural SPL Testing. *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '16*, ACM Press: Salvador, Brazil, 2016; 89–96, doi:10.1145/2866614.2866627.

49. Vargha A, Delaney HD. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 2000; **25**(2):101–132.

50. Arcuri A, Fraser G. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* jun 2013; **18**(3):594–623, doi:10.1007/s10664-013-9249-9.

51. Arcuri A, Briand L. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 2014; **24**(3):219–250, doi:10.1002/stvr.1486.

52. Panichella A, Molina UR. Java unit testing tool competition - Fifth round. *Proceedings - 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing, SBST 2017* 2017; :32–38doi:10.1109/SBST.2017.7.