Enabling Efficient Assertion Inference

Aayush Garg*, Renzo Degiovanni*, Facundo Molina†, Maxime Cordy*
Nazareno Aguirre‡, Mike Papadakis* and Yves Le Traon*
*University of Luxembourg, Luxembourg
†IMDEA Software Institute, Spain
‡University of Río Cuarto and CONICET, Argentina
aayush.garg@uni.lu, renzo.degiovanni@uni.lu, facundo.molina@imdea.org, maxime.cordy@uni.lu,
naguirre@dc.exa.unrc.edu.ar, michail.papadakis@uni.lu, yves.letraon@uni.lu

***Abstract*—Assertion inference techniques aim at automatically inferring sets of program assertions that capture the exhibited software behavior, often by generating and filtering assertions through dynamic test executions and mutation testing. Although powerful, such techniques are computationally expensive due to the large number of mutants that require execution. In this study, we introduce the notion of *Assertion Inferring Mutants*, and demonstrate that these mutants are sufficient for assertion inference and correspond to a small subset (12.95%) of the entire mutant set. Moreover, these mutants are significantly different (71.59%) from *Subsuming Mutants* that are frequently cited by mutation testing literature. We also show that *Assertion Inferring Mutants* can be statically approximated via a learning-based method. Given the widespread adoption of encoder-decoder architecture for prediction tasks, we demonstrate that it predicts *Assertion Inferring Mutants* with 0.79 Precision and 0.49 Recall. Its evaluation on 46 projects showcases that it enables a comparable inference capability (missing only 12.49% assertions) with a complete mutation analysis, while significantly reducing the execution cost (achieving 46.29 times faster inference). Moreover, it enables assertion inference techniques to scale on subjects where complete mutation testing is prohibitively expensive and other mutant selection strategies do not lead to an acceptable assertion inference.**

I. Introduction

Software specifications aim at describing the software's intended behavior, and can be used to distinguish correct from incorrect software behaviors. While these are typically described informally (e.g., via API documentation), specifications become significantly more useful when expressed formally as executable constraints/specifications. Executable specifications are typically expressed as code/program assertions for various program points, such as method preconditions and postconditions, that must hold true at the corresponding program points during execution. Program assertions are known to be useful in many software engineering tasks, e.g., test generation [15], [51], bug finding [33], [40] and automated debugging [16], [34], [44]. However, they are tedious to write and maintain, and as a result developers often elude providing them [8], [55].

To address this issue, different techniques that automatically infer assertions for specific program points have been proposed [37], [38], [50]. These techniques generate candidate assertions, and use dynamic test executions to determine which assertions are consistent with the behavior exhibited by a provided test suite, and mutation testing to discard ineffective/weak assertions that are unable to detect artificially seeded faults (mutants), i.e., assertions that are never falsified during mutants' execution. Though powerful, these techniques are computationally expensive due to the large number of assertions to analyze, and the large numbers of tests and mutants that have to be executed. The problem is further escalated when working with large programs, as the number of mutants grows proportionally to the program size. For instance, the state of the art technique SpecFuzzer [37] times out (requires more than 90 minutes to run) in programs with 180 lines of code.

To reduce the computational demands, it is imperative to limit the number of mutants involved (fewer mutants result in fewer executions). Interestingly, we find that the majority of the mutants used by the existing assertion inference techniques are redundant. This implies that discarding these mutants does not impact the quality of inferred assertions. We thus introduce the notion of *Assertion Inferring Mutants*, the subset of mutants produced by a mutation testing tool that is sufficient to effectively identify relevant candidate assertions (i.e., the assertions that fail at least once on mutants).

We demonstrate that *Assertion Inferring Mutants* represent 12.95% of the mutants generated by Major [28] (the mutation testing tool employed in previous studies), allowing for drastic assertion inference overhead reductions. At the same time, *Assertion Inferring Mutants* are significantly different from *Subsuming Mutants* (which have been studied in the literature [23], [42] and have been shown to improve efficiency) with 71.59% of *Assertion Inferring Mutants* not being subsuming. This implies that subsuming mutant selection techniques are ineffective for assertion inference, as they would miss many assertions (48.53% assertions missed, according to our results).

Thus, we explore learning-based approach to statically identify *Assertion Inferring Mutants* given their contextual information. In particular, given the widespread adoption of encoder-decoder architecture [29] that has been established to accomplish many software engineering tasks [5], [21]–[23], [49], [54], we employ it to learn the associations between mutants and their surrounding code with respect to the assertion inference task. This implies that our learning scope is the area around the mutation point that locally identifies the mutants that are most likely to be useful for assertion inference. We follow the architecture design proposed by the

previous studies and refer to it as *Seeker*[1] throughout this paper. It operates at the lexical level, with a simple code pre-processing that represents mutants and their surrounding code as vectors of tokens with all user-defined identifiers (e.g., variable names) replaced by predefined and predictable identifier names. This representation allows us to restrict the learning scope to a relatively small number of tokens around the mutation points enabling inter-project predictions. We train the encoder-decoder architecture on code fragments and extract code embeddings. These embeddings are learned with corresponding labels using a classifier [9] to enable prediction.

We evaluate *Seeker*'s ability to predict *Assertion Inferring Mutants* on a set of 46 programs, composed of 40 programs taken from previous studies [37], [38], [50] and 6 large Maven projects taken from GitHub, to evaluate scalability. Our results demonstrate that *Seeker* can statically select *Assertion Inferring Mutants* with 0.79 Precision and 0.49 Recall, overall yielding 0.58 MCC[2].

Surprisingly, by performing assertion inference based only on *Seeker*'s predicted mutants (instead of all mutants), we reduce assertion inference time (wall clock) by 46.29 times with only 12.49% assertion missed. Additionally, when comparing with randomly selected sets of mutants (same number as those selected by *Seeker*), we observe a clear advantage of *Seeker* in terms of effectiveness, i.e., *Seeker* enables inference of 36% more assertions while consuming an approximately equal amount of execution time as *Random Mutant Selection*.

More importantly, since *Seeker* selects a few mutants, it enables assertion inference technique SpecFuzzer to scale by allowing its operation on our considered 6 real-world subjects, where a complete mutation testing is prohibitively expensive. In half of these subjects, *Random Mutant Selection* does not lead to any assertion inference and is subsumed by *Seeker* in the other half of the subjects.

II. Background & Related Work

A. Specification Inference

Software specifications are descriptions of the intended behavior of software. They are crucial for determining if software behavior is correct. The provision of software specifications is strongly related to the oracle problem, i.e., the problem, in the context of software testing, of determining whether the results of program executions are coherent with the desired behavior of the program [7]. Though specifications are typically expressed informally (e.g., via API documentations), when these are expressed more formally as a set of executable constraints/assertions, they have powerful applications in many software engineering tasks such as software design [36], software testing [4], [19], and verification [14], [20].

[1]The name *Seeker* comes from a seeker's role to search for and catch the Golden Snitch, in the fictional sport of Quidditch invented by the author *J.K. Rowling* for her fantasy book series *Harry Potter* [46]. In the context of our study, *Seeker* searches for (seeks) *Assertion Inferring Mutants*.

[2]*Matthews Correlation Coefficient* (MCC) [56] is a reliable metric of the quality of prediction models [47], relevant when the classes are of different sizes, e.g., *12.95% Assertion Inferring Mutants* in total (in comparison to 87.05% low utility mutants), for subjects in our dataset.

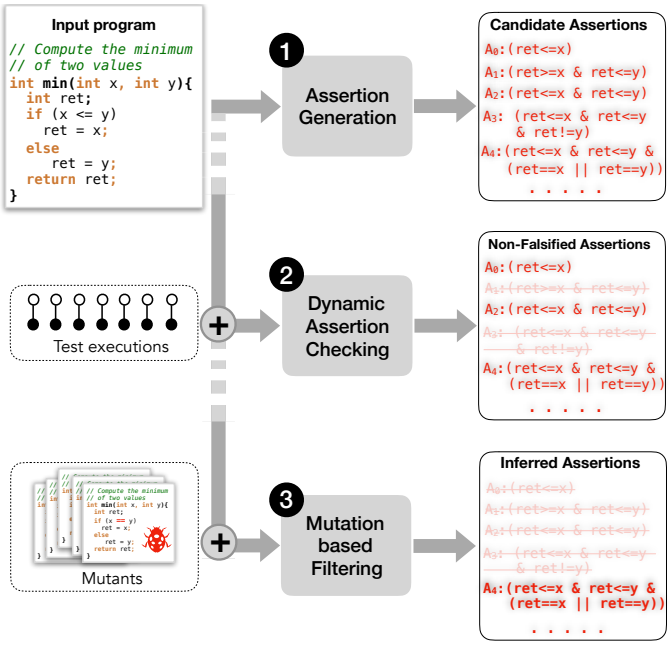

Fig. 1. Assertion Inference via Dynamic Test Execution & Mutation Analysis

The specification inference problem consist of automatically generating specifications from existing software artifacts, e.g., documentation, source code, program executions, etc. At the source code level, formal program specifications are composed of a set of (executable) assertions for various program points, such as method preconditions, postconditions and invariants, that must hold true during program execution, at the corresponding program points. In this paper, we focus on *postcondition* assertions, i.e., assertions that state the properties that are expected to hold after a given method is executed.

Figure 1 depicts the typical process of existing assertion inference techniques [37], [38], [50]. First, the assertion generation step, based in general on a search-based algorithm (e.g. GAssert [50] and EvoSpex [38] use evolutionary search algorithms, while SpecFuzzer [37] uses fuzzing), produces a set of candidate assertions for a given program/method. Second, the program's test suite (given as input or automatically generated) is executed to determine which of those assertions are coherent with the behaviors currently exhibited by the program. Lastly, the non-falsified assertions (i.e., those that are coherent with the test suite executions) go through a mutation analysis step for filtering out weak assertions. Here, a non-falsified assertion that is also coherent with all the mutants' execution of a given program, is considered to be weak because it is unable to distinguish between the original and the mutated program behaviors, and is hence discarded. The inferred assertions are the ones that are coherent with the current program behavior but are falsified by the behavior of buggy programs (i.e., they kill at least one mutant).

Modern assertion inference techniques take on *Daikon* [17], a well-known dynamic technique that infers assertions by monitoring test executions. Given a program under analysis and a test suite, Daikon execute the tests, monitors the program states at various points, and then evaluates candidate

assertions, obtained by instantiating assertion patterns on the program states. Those assertions that are *never falsified* by any test at a given program point are reported as likely invariants at the program point. As Daikon does not use mutation analysis or any other sophisticated mechanism to detect irrelevant/redundant assertions, it often report many assertions that can be weak or redundant with respect to other reported program assertions [37].

GAssert [50] and *EvoSpex* [38] are assertion inference techniques based on evolutionary search algorithms. Similar to Daikon, these tools execute a test suite of the program under analysis and observe the execution in order to infer assertions that are consistent with the observations. By favoring shorter assertions during evolution, and also favoring assertions that are able to detect buggy behaviors via mutation analysis, these techniques are able infer shorter and stronger assertions, compared to Daikon. However, as the components of the evolutionary process are specifically designed to handle the assertion languages these tools support, changing or extending these languages implies redefining evolutionary operators and other elements of the process, which is non-trivial.

SpecFuzzer [37] is another assertion inference technique which infers assertions through a combination of static analysis, grammar-based fuzzing, and mutation analysis. First, it uses a lightweight static analysis to produce a grammar for the assertion language, which is tuned to the program under analysis. Second, it uses a grammar-based fuzzer to generate candidate assertions from the grammar. Then, a dynamic detector determines which of those assertions are consistent with the behavior exhibited by a provided test suite. Finally, SpecFuzzer eliminates redundant and irrelevant assertions using a selection mechanism based on mutation analysis. A salient feature of SpecFuzzer is that developers can adjust the produced specifications by tuning the grammar, as opposed to making changes to a search algorithm, as GAssert and EvoSpex would require.

It is worth remarking that all of the above described techniques infer assertions from the current program behavior, which may not necessarily be the intended program behavior, if the program is incorrect. Inferred assertions are useful for many tasks, including regression and differential analyses, as well as for program understanding.

B. Assertion Inferring Mutants

Steps 2 and 3 from Figure 1 show that the generated *candidate assertions* undergo a two-step filtering process. In step 2, assertions that are falsified when running the test suite of a target class *C* are discarded, since these are invalid assertions not satisfying the legit program behaviour exhibited by the test suite execution. Though important to identify *valid assertions*, such filtering is not enough as it leaves room for *weak assertions*, i.e, assertions that are trivial to satisfy and would not trigger any error if the target class *C* had any incorrect behaviour. For instance, a tautology such as $assert(x >= y \mid\mid x <= y)$ is a valid assertion that cannot be falsified, but it is unlikely to be useful. In the case of SpecFuzzer [37], the fuzzer reports thousands of candidate assertions, and only a few are falsified by the test suite.

Such weak assertions are not useful and thus the use of mutation analysis has been proposed to identify and discard them (step 3) [37], [38], [50]. The underlying idea is that valid assertions that are also coherent with every mutant's execution of target class *C* are weak because they represent properties that hold also for buggy versions of *C* (the mutants). On the contrary, assertions that do not hold for at least one mutant of *C*, are useful because they are capable of distinguishing buggy versions of the code. Given a target class *C* and a set A of candidate assertions that are consistent with the behavior of *C*, a mutant *C'* of *C* is called *assertion inferring* if at least one assertion in A is able to kill of *C'*.

Despite being effective for discarding weak assertions, mutation analysis suffers from scalability issues due to the large number of mutants that can be generated from even a small piece of code. This adversely affects the overall performance of assertion inference techniques, especially on large subjects. Our goal in this paper is to effectively identify *Assertion Inferring Mutants* to improve the scalability of assertion inference techniques.

C. Mutant Selection

Mutation analysis is computationally expensive even beyond its use for assertion inference. This is mainly due to the large number of mutants that it introduces, all of which require analysis and execution. To reduce its application cost, it is imperative to limit the number of mutants to those that are actually useful, prior to any manual mutant analysis or test execution. This problem is known as the mutant selection problem [43] and has been studied in the form of selective mutation [39], [57], i.e., restricting the number of transformations to be used, with limited success [11], [31]. A main issue with selective mutation is the simple syntactic-based nature of the selection process: a restricted set of transformations is applied in every appropriate program location, thus ignoring the program semantics and the contexts of the mutated locations.

D. Subsuming Mutants

In traditional mutation testing – where the goal is to assess the ability of a test suite in "killing" mutants (i.e., distinguishing the observable behaviors of the mutant and the original program) – one can reduce the number of mutants to be analyzed by identifying the *subsuming mutants* [3], [23], [30]. Given two mutants M_1 and M_2, M_1 subsumes M_2 if every test case T killing M_1 also kills M_2. The cost of mutation analysis can then be reduced by identifying the minimal subset of subsuming mutants, such that any test suite able to kill these mutants can also kill the entire set of killable mutants (excluding mutants that are functionally equivalent to the original program and cannot be killed). Hence, practitioners can perform mutation testing efficiently by analyzing only subsuming mutants.

Given the potential of subsuming mutants in reducing mutation testing overhead, it is reasonable to investigate if they

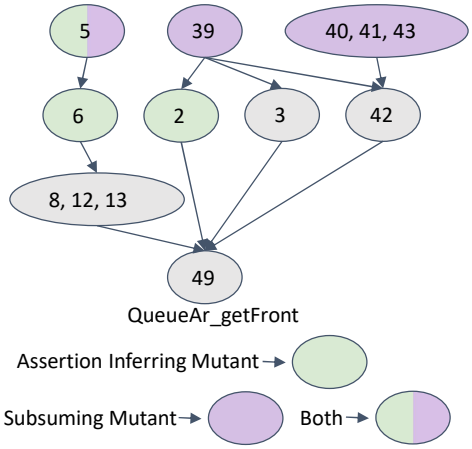

Fig. 2. Mutant subsumption hierarchy for the subject `QueueAr_getFront` showing the positions of *Assertion Inferring Mutants* and *Subsuming Mutants*

are also suitable for assertion inference, i.e., if they can help to more efficiently discard weak assertions. As we discuss in Section VII-A, subsuming mutants are generally not sufficient for the assertion inference task as their use results in losing almost half of the inferred assertions, compared to considering all mutants.

III. Illustrative Example

Figure 2 shows the mutants generated for the method `getFront()` of class `QueueAr`, one of our subjects. The graph depicts the mutants' subsumption hierarchy, which is a standard way of representing subsumption relations between a set of mutants generated from a given subject. Nodes represent mutants of the subject, and an edge connecting mutant M_1 to mutant M_2 represents the fact that M_2 is subsumed by M_1. In our example, mutant 39 subsumes mutants 2, 3 and 42. Mutually subsuming mutants are typically merged into a single node – e.g., mutants 40, 41 and 43 are mutually subsuming. Our figure highlights in purple the subsuming mutants (those at the top of the hierarchy), and in green the *Assertion Inferring Mutants*.

To analyze the impact that mutation analysis has in the inference process, we first inferred assertions with SpecFuzzer [37] on the subject `QueueAr_getFront` with its default configuration, i.e., using all available mutants. SpecFuzzer inferred *27* assertions, with the assertion filtering step via mutation analysis (step 3 of Figure 1) taking 91 minutes on our infrastructure (see Section VI). By contrast, if we only use subsuming mutants in the filtering step, it only takes 2.5 minutes (36.4 times faster), but produces just 5 assertions. These results evidence that, while reducing the number of mutants to analyze can improve the computational efficiency of the filtering process, subsuming mutants are not appropriate for this task. Intuitively, this is because the initial purpose of subsuming mutants is to minimize the number of tests needed to kill all mutants. In the context of assertion inference one aims instead at inferring all valid assertions that can distinguish the mutants from the original code, that is, generate as many assertions that capture the specific code properties. For instance, in our `QueueAr_getFront` example, 5 out of the 27 inferred assertions are falsified when executing mutant *5*. On the other hand, mutant *6* helps in inferring 21 assertions (i.e., 21 out of the 27 assertions are falsified during mutant *6* execution); while mutant *2* helps in inferring the remaining assertion. In other words, by considering only the five subsuming mutants (i.e., mutants *5*, 39, 40, 41 and 43), and discarding subsumed mutants (including mutants *6* and *2*), the assertion inference results in reporting only 5 assertions, losing 22 strong assertions that could have been inferred by using just the three *Assertion Inferring Mutants* (or the entire pool of mutants at the expense of a significantly higher computational cost).

The above example demonstrates the difference between *Subsuming Mutants* and *Assertion Inferring Mutants*, and the need for an approach that can efficiently identify the latter in order to save valuable time on the mutation analysis step, while maintaining the benefits of assertion inference. The *Seeker* technique that we propose in this paper is the first mutant selection method especially designed for predicting *Assertion Inferring Mutants*, making existing specification inference techniques more efficient and scalable. As an example, on the *QueueAr_getFront* example, *Seeker* predicts mutant *6* as assertion inferring mutant and helps SpecFuzzer to infer 21 assertions (out of 27 assertions when using all mutants), using only a fraction of the computation time (30 seconds) that analyzing all mutants requires (91 minutes).

IV. Approach

The main objective of *Seeker* is to predict whether a mutant (of a previously unseen piece of code) is likely to be assertion inferring. To make our approach lightweight in terms of engineering and computational effort, we want *Seeker* to be able to (a) learn relevant features of *Assertion Inferring Mutants* without requiring manual feature definition, and (b) do so without costly dynamic analysis of mutant executions. To achieve this, we decompose our problem into two parts: learn a representation of mutants using code embedding techniques, and learn to predict, based on such embeddings, whether the represented mutants are *Assertion Inferring Mutants*.

A. *Overview of* Seeker

Figure 3 shows an overview of *Seeker*. We decompose our approach into three steps that we detail later on in this section:

1) *Build a token representation*: *Seeker* pre-processes the original code in order to remove irrelevant information and produces abstracted code, which is then tokenized to form a sequence of tokens. Each mutant is ultimately transformed into its corresponding token representation and undergoes the next step.
2) *Representation learning*: We train an encoder-decoder model to generate an embedding, aka vector representation of the mutant. This step is where *Seeker* automatically learns the relevant features of mutants without requiring an explicit definition of these features.

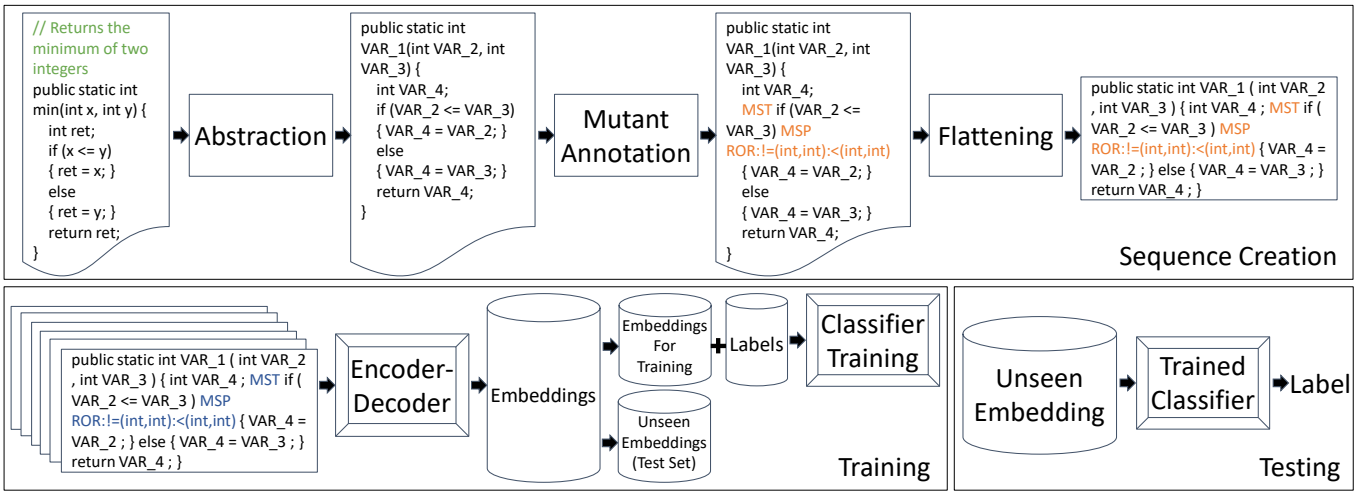

Fig. 3. Overview of *Seeker*: Source code is abstracted and annotated to represent a mutant, which is further flattened to create a space separated sequence of tokens. An encoder-decoder model is trained on token sequences to generate mutant embeddings. A classifier is trained on these embeddings and their corresponding labels (whether or not the mutant is assertion inferring). The trained classifier can then be used for label prediction of an unseen mutant.

3) *Classification: Seeker* trains a classification model to classify the mutants (based on their embeddings) as *Assertion Inferring Mutants* or not. The true labels used for training are obtained by running SpecFuzzer on the original code, and checking whether the mutants are *Assertion Inferring Mutants* (i.e., which mutants are killed only by assertions coherent with the test-suite).

It is interesting to note that the mutant representation learned by *Seeker* does not depend on the particular set of assertions that SpecFuzzer (or any other assertion inference technique) would check against the mutant. *Seeker* aims instead at learning properties of the mutants (and their surrounding contexts) that are generally useful for assertion inference. This is in line with the recent work on contextual mutant selection [12], [23], [27] that aims at selecting high utility mutants for mutation testing. This characteristic makes *Seeker* applicable to pieces of code that have not been seen during training. In particular, our experiments reveal the ability of *Seeker* to be effective on projects not seen during training.

The assertion inference technique that is used to build the true labels in the classification task is very important as this technique should produce assertions that capture the software behavior as precisely as possible in order to distinguish the buggy versions of the code, i.e., mutants. This is an essential condition for our classifier to provide relevant prediction results. In our study, we employ SpecFuzzer [37] that has been shown to outperform related techniques (i.e., GAssert [50] and EvoSpex [38]) in assertion inference by inferring 7 times more assertions than GAssert, and 15 times more assertions than EvoSpex. Simultaneously, it has been shown to achieve better performance (F-1 score) than the related approaches for producing developer-validated assertions.

B. Training Sequences Generation

A major challenge in learning from raw source code is the huge vocabulary created by the abundance of identifiers and literals used in the code [2], [52], [53]. In our case, this large vocabulary may hinder *Seeker*'s ability to learn relevant features of *Assertion Inferring Mutants*. Thus, we first abstract original (non-mutated) source code by replacing user-defined entities (function names, variable names, and string literals) with generic identifiers that can be reused across the source code file. During this step, we also remove code comments. This pre-processing yields an abstracted version of the original source code, as the abstracted code snippet in Figure 3.

To perform the abstraction, we use the publicly available tool *src2abs* [52]. This tool first discerns the type of each identifier and literal in the source code. Then, it replaces each identifier and literal in the stream of tokens with a unique ID representing the type and role of the identifier/literal in the code. Each ID `<TYPE>_#` is formed by a prefix, (i.e., `<TYPE>_`) which represents the type and role of the identifier/literal, and a numerical ID, (i.e., #) which is assigned sequentially when reading the code. These IDs are reused when the same identifier/literal appears again in the stream of tokens. Although we use src2abs, one can use any other utility that identifies user-defined entities and replaces such with reusable identifiers.

Next, to represent a mutant, we annotate the abstracted code with a mutation annotation on the statement where the mutation is to be applied. These annotations have the general shape "`MST` statement `MSP MutationOperator`", where `MST` and `MSP` denote mutation annotation start and stop, respectively, and are followed by a `MutationOperator` to indicate the applied mutation operation (as shown in figure 3). We repeat the process for every mutant.

Finally, we flatten every mutant (by removing newline, tabs and extra whitespace) to create a single space separated sequence of tokens. Using these sequences, we intend to capture as much code as possible around the mutant without incurring in a prohibitively expensive training time [21]–[23], [52], [54]. We found a sequence length of 500 tokens to be a good fit for our task as it does not exceed 24 hours of training time (wall clock) on a Tesla V100 GPU.

C. Embedding Learning with Encoder-Decoder

Our next step is to learn embeddings, aka vector representations, from mutants' token representation that can later on be used to train a classification model. We develop an encoder-decoder model, a neural architecture commonly used in representation learning tasks [29]. The key principles of our encoder-decoder architecture are that the encoder transforms the token representation into an embedding and the decoder attempts to retrieve the original token representation from the encoded embedding. The learning objective is then to minimize the binary cross-entropy between the original token representation and the decoded one. Once the model training has converged, we can compute the embedding from any other mutant's token representation by feeding the latter into the encoder and retrieving the output.

We use a bi-directional Recurrent Neural Network (RNN) [10] to develop our encoder-decoder, as previous works on code learning have demonstrated the effectiveness of these models to learn useful representations from code sequences [5], [21]–[23], [49]. We build *Seeker* on top of *tf-seq2seq* [1], an established general-purpose encoder-decoder framework. We use a Gated Recurrent Units (GRU) network [13] to act as the RNN cell, which was shown to perform better than simpler alternatives (e.g. simple RNNs) both in software engineering and other learning tasks [23], [48]. To achieve good performance with acceptable model training time, we utilize AttentionLayerBahdanau [6] as our attention class, configured with 2 layered AttentionDecoder and 1 layered BidirectionalRNNEncoder, both with 256 units.

To determine the number of training epochs for model convergence, we conducted a preliminary study involving a small validation set (independent of both the training and test sets used in our evaluation) where we monitor the model's performance in replicating (as output) the same mutant sequence provided as input. We pursue training the model until the training performance on the validation set does not further improve. We found 10 epochs for the sequences up to a length of 500 tokens to be a good default for our validation sets.

D. *Classifying* Assertion Inferring Mutants

Next, we train a classification model in predicting whether a mutant (represented through the embedding produced by the RNN encoder) is likely to be an assertion inferring mutant. The learning objective here is to maximize the classification performance (which we mainly measure with Matthews Correlation Coefficient (MCC), see Section VI-B). To obtain our true classification labels, we run an assertion inference technique (viz. SpecFuzzer) using all available mutants and exhaustively determined which mutants are assertion inferring. As for the classification model, we rely on *random forests* [9] because these are lightweight to train and have shown to be effective in solving various software engineering tasks [26], [45]. We used standard parameters for random forests, viz. we set the number of trees to 100, use Gini impurity for splitting, and set the number of features (i.e., embedding logits) to consider at each split to the square root of the total number of features.

Once the model training has converged, we can use the random forest to predict whether an unseen mutant is likely to be assertion inferring. For the actual classification, we make the mutant go through the pre-processing pipeline to obtain its abstract token representation, then feed it into the encoder-decoder architecture to retrieve its embedding and finally input it into the classifier to obtain the predicted label (assertion inferring or not).

V. Research Questions

We start our analysis by investigating the prediction performance of *Seeker* to select *Assertion Inferring Mutants* and compare whether these can be approximated by other sets of mutants, namely, *Subsuming Mutants*. Thus, we ask:

RQ1 *Prediction Evaluation:* How effective is *Seeker* in predicting *Assertion Inferring Mutants*? Can subsuming mutants approximate them?

To determine which mutants are assertion inferring (i.e. those killed by at least one assertion), we consider the dataset provided by *Molina et al.* [37] and execute the state of the art assertion inference technique SpecFuzzer on 40 subjects without discarding any mutant. Then, we analyze the performance of *Seeker* in identifying these mutants. We compare the results with the set of subsuming mutants since they form the main objective of mutant selection [23], [32], [41] with numerous strategies targeting them [23], [24], [27], [35].

Since *Seeker*'s predictions might not be perfect, we also assess its performance in the context of assertion inference, and contrast it with other mutant selection strategies, namely, *random mutant* selection and *subsuming mutants*. We consider random mutant selection since it is an untargeted method that is often superior to many mutant selection strategies [25], [58] and is considered by the literature as a strong baseline [12], [23], [32]. Hence, we check the effectiveness (completeness w.r.t. to using all mutants) and efficiency (how much time is required) of SpecFuzzer [37] when utilizing the different mutant subsets over all supported mutants. Therefore, we ask:

RQ2 *Inference Evaluation:* How effective and efficient is *Seeker* in comparison to subsuming, randomly selected and all mutants baseline methods with respect to the assertion inference task?

For this task, we re-execute SpecFuzzer on the 40 subjects, by selecting the mutants following *Seeker* and our two baseline mutant selection techniques (subsuming and random mutant selection), and compare its performance when executing SpecFuzzer without discarding any mutant.

Finally, in order to investigate if *Seeker*'s predicted mutants can help the assertion inference technique SpecFuzzer to scale, additional subjects (other than the subjects considered by *Molina et al.* [37]) must be taken into account. Here, we determine if considering only *Seeker*'s predicted mutants can aid SpecFuzzer to infer assertions in cases where SpecFuzzer times out if all mutants are considered. Thus, we conduct experiments on 6 large subjects from GitHub (Table I) where

SpecFuzzer timed out. We also compare SpecFuzzer's performance when it considers *Seeker*'s predicted mutants vs an equal number of randomly selected mutants. Hence, we ask:

RQ3 *Scalability Evaluation:* How does the inclusion of *Seeker* in assertion inference techniques impact scalability?

VI. Experimental Setup

A. Data and Tools

We select 46 Java methods; 40 subjects used in previous studies [37], [38], [50] for evaluating *Seeker*'s performance in RQ1 and RQ2, and 6 larger subjects from GitHub for the scalability evaluation in RQ3.

Table I records the details of our dataset. For each method analyzed, it reports the total number of mutants generated, the number of *Assertion Inferring Mutants*, and the total number of assertions inferred when considering all mutants (i.e., without mutant selection).

To perform mutation testing we use Major [28], and to construct comprehensive test suites (and improve the chances to infer true assertions), we use EvoSuite [18] and Randoop [40] to augment the developer test suites, similarly to what was done by previous work [37].

B. Prediction Performance Metrics

Seeker's predictions can result in four types of outputs. Given a mutant that is assertion inferring, if it is predicted as assertion inferring, then it is a true positive (TP); otherwise, it is a false negative (FN). Vice-versa, if a mutant that does not infer any assertion is predicted as assertion inferring, then it is a false positive (FP); otherwise, it is a true negative (TN). We can then compute the traditional evaluation metrics such as *Precision* and *Recall*, which quantitatively evaluate the prediction accuracy of prediction models.Intuitively, *Precision* indicates the ratio of correctly predicted positives over all the considered positives. *Recall* indicates the ratio of correctly predicted positives over all actual positives. Yet, these metrics do not take into account the true negatives and can be misleading, especially in the case of imbalanced data. Hence, we complement these with the *Matthews Correlation Coefficient (MCC)*, a reliable metric of the quality of prediction models [56]. It is regarded as a balanced measure that can be used even when the classes are of very different sizes [47], as in our case, where we have *12.95% Assertion Inferring Mutants* in total, for the 40 subjects in the dataset (Table I). *MCC* is calculated as:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$$

MCC returns a coefficient between 1 and -1. An MCC value of 1 indicates a perfect prediction, while a value of -1 indicates a perfect inverse prediction, i.e., a total disagreement between prediction and reality. MCC value of 0 indicates that the prediction performance is equivalent to random guessing.

C. Experimental Procedure

To answer our RQs we execute SpecFuzzer to infer assertions for all subjects (Table I) with its default setup, i.e., using all mutants to filter candidate assertions during the mutation analysis step (Figure 1). We also determine *Assertion Inferring Mutants* and *Subsuming Mutants* from SpecFuzzer execution logs for the 40 subjects used in RQ1 and RQ2. Once the mutants are labeled, we re-execute SpecFuzzer by employing the following 3 mutant selection techniques:

- *Subsuming Mutant Selection*. We execute SpecFuzzer by only considering subsuming mutants for mutation analysis.
- *Seeker*. We train models on *Assertion Inferring Mutants* and perform k-fold cross validation (where k = 5) at the project level, i.e., we train on 32 subjects and evaluate/test on 8 unseen during testing subjects, and repeat 5 times. Once we get the predictions for all 40 subjects, we re-execute SpecFuzzer by only considering the mutants predicted as assertion inferring.
- *Random Mutant Selection*. We randomly select an equal number of mutants (equal to the number of mutants predicted as assertion inferring) from the original set of mutants and re-execute SpecFuzzer by only considering these randomly selected mutants. We repeat this step 10 times to eliminate the chances to report coincidental results. We report the median case results.

To answer *RQ1*, we compute the Prediction Performance Metrics of *Seeker* in order to show its learning ability. This is a sanity check that our prediction modeling framework indeed manages to predict something well. However, prediction results do not reflect the end-task (assertion inference) performance since mutants are not independent, there are large overlaps between the tests and assertions that lead to mutant kills. To answer *RQ2*, we thus measure the cost of the employed mutant selection technique, i.e., how many of the assertions inferred when all mutants are considered, are *not* inferred when mutant selection is used, and the benefit gained, i.e., the improvement in terms of wall clock time.

To answer *RQ3*, i.e., if *Seeker*'s predicted mutants can help SpecFuzzer to infer assertions for 6 subjects where it was not able to infer any assertion (timed out when all mutants were considered for analysis), we retrain *Seeker* on all 40 subjects (with available labeled mutants) and predict likely *Assertion Inferring Mutants* for these 6 subjects. We re-execute SpecFuzzer by only using the predicted mutants and by discarding all other mutants from the original set. Additionally, we randomly select mutants in a similar fashion as before (following RQ2 experimental procedure) and re-execute SpecFuzzer accordingly to compare performance with *Random Mutant Selection*. Thus to answer *RQ3* we measure – 1) In how many subjects, the selected mutants lead to assertion inference, and 2) The ratio of assertion inferring mutants from the entire set of mutants.

TABLE I

THE TABLE RECORDS THE TEST SUBJECTS, METHOD DETAILS, ALL MUTANTS COUNT, ASSERTION INFERRING MUTANTS COUNT, AND INFERRED ASSERTIONS COUNT WHEN ALL MUTANTS ARE CONSIDERED, (I.E., SPECFUZZER'S DEFAULT EXECUTION WITH NO MUTANT SELECTION). ADDITIONALLY, THE TABLE RECORDS *Seeker*'S PREDICTION PERFORMANCE SCORES FOR EVERY PROJECT.

Subject	Method	All Mutants	Assertion Inferring Mutants	All Inferred Assertions	*Seeker*'s Prediction Performance						
					TP	TN	FP	FN	Precision	Recall	MCC
ArithmeticUtils_subAndCheck	math.ArithmeticsUtils.subAndCheck	16	2	3	1	14	0	1	1.0	0.5	0.68
BooleanUtils_compare	lang.BooleanUtils.compare	13	13	29	13	0	0	0	1.0	1.0	1.0
composite_addChild	eiffel.Composte.addChild	35	6	185	6	28	1	0	0.86	1.0	1.0
doublylinkedlistnode_insertRight	eiffel.DLLN.insert_right	18	7	16	1	11	0	6	1.0	0.14	0.3
doublylinkedlistnode_remove	eiffel.DLLN.remove	18	4	21	1	14	0	3	1.0	0.25	0.45
Envelope_maxExtent	tsuite.Envelope.maxExtent	56	10	188	6	45	1	4	0.86	0.6	0.67
FastMathNew_floor	math.FastMath.floor	42	18	60	1	24	0	17	1.0	0.06	0.18
IntMath_mod	guava.IntMath.mod	21	15	199	12	5	1	3	0.92	0.8	0.62
listcomp02_insert_r	cozy.ListComp02.insert_r	20	2	1	1	18	0	1	1.0	0.5	0.69
listcomp02_insert_s	cozy.ListComp02.insert_s	20	1	1	1	19	0	0	1.0	1.0	1.0
map_count	eiffel.Map.count	63	3	4	1	60	0	2	1.0	0.33	0.57
map_extend	eiffel.Map.extend	65	9	10	1	56	0	8	1.0	0.11	0.31
map_remove	eiffel.Map.remove	63	1	1	1	62	0	0	1.0	1.0	1.0
MathUtilsNew_copySignInt	math.MathUtils.copySignInt	48	2	16	1	46	0	1	1.0	0.5	0.7
MathUtil_clamp	tsuite.MathUtil.clamp	11	8	12	8	2	1	0	0.89	1.0	1.0
maxbag_add	cozy.MaxBag.add	748	53	49	1	695	0	52	1.0	0.02	0.13
maxbag_getMax	cozy.MaxBag.get_max	749	21	25	19	727	1	2	0.95	0.9	0.93
maxbag_remove	cozy.MaxBag.remove	748	67	26	44	680	1	23	0.98	0.66	0.79
polyupdate_a1	cozy.PolyUpdate.a	54	26	100	26	27	1	0	0.96	1.0	1.0
polyupdate_sm	cozy.PolyUpdate.sm	56	13	73	13	42	1	0	0.93	1.0	1.0
QueueAr_dequeue	daikon.QueueAr.dequeue	66	9	68	1	57	0	8	1.0	0.11	0.31
QueueAr_dequeueAll	daikon.QueueAr.dequeueAll	67	11	69	1	56	0	10	1.0	0.09	0.28
QueueAr_enqueue	daikon.QueueAr.enqueue	66	17	119	1	49	0	16	1.0	0.06	0.21
QueueAr_getFront	daikon.QueueAr.getFront	67	3	27	1	62	0	4	1.0	0.2	0.43
QueueAr_makeEmpty	daikon.QueueAr.makeEmpty	67	20	73	1	47	0	19	1.0	0.05	0.19
ringbuffer_count	eiffel.RingBuffer.count	101	28	119	15	72	1	13	0.94	0.54	0.64
ringbuffer_extend	eiffel.RingBuffer.extend	101	20	148	5	80	1	15	0.83	0.25	0.4
ringbuffer_item	eiffel.RingBuffer.item	101	11	116	11	89	1	0	0.92	1.0	1.0
ringbuffer_remove	eiffel.RingBuffer.remove	101	14	143	14	86	1	0	0.93	1.0	1.0
ringbuffer_wipeOut	eiffel.RingBuffer.wipe_out	101	13	95	13	87	1	0	0.93	1.0	1.0
simple-examples_abs	oasis.SimpleMethods.abs	20	18	30	18	1	1	0	0.95	1.0	1.0
simple-examples_addElementToSet	oasis.SimpleMethods.addElementToSet	3	2	1	2	1	0	0	1.0	1.0	1.0
simple-examples_getMin	oasis.SimpleMethods.getMin	7	6	51	1	1	0	5	1.0	0.17	0.17
StackAr_makeEmpty	daikon.StackAr.makeEmpty	47	13	47	1	34	0	12	1.0	0.08	0.24
StackAr_pop	daikon.StackAr.pop	63	10	35	1	53	0	9	1.0	0.1	0.29
StackAr_push	daikon.StackAr.push	55	6	25	1	49	0	5	1.0	0.17	0.39
StackAr_top	daikon.StackAr.top	50	8	3	1	42	0	7	1.0	0.12	0.33
StackAr_topAndPop	daikon.StackAr.topAndPop	54	13	68	1	41	0	12	1.0	0.08	0.24
structure_foo	cozy.Structure.foo	27	5	1	1	22	1	4	0.5	0.2	0.23
structure_setX	cozy.Structure.setX	26	15	131	10	11	0	5	1.0	0.67	0.68
EmailScanner_findFirst	nibor.autolink.internal.EmailScanner.findFirst	134	Subject considered for Scalability Evaluation (RQ3)* — SpecFuzzer timed out								
EmailScanner_scan	nibor.autolink.internal.EmailScanner.scan	134	Subject considered for Scalability Evaluation (RQ3)* — SpecFuzzer timed out								
IdentityHashSet_isEmpty	leplus.ristretto.util.IdentityHashSet.isEmpty	23	Subject considered for Scalability Evaluation (RQ3)* — SpecFuzzer timed out								
OptionGroup_setRequired	apache.commons.cli.OptionGroup.setRequired	34	Subject considered for Scalability Evaluation (RQ3)* — SpecFuzzer timed out								
OptionGroup_setSelected	apache.commons.cli.OptionGroup.setSelected	34	Subject considered for Scalability Evaluation (RQ3)* — SpecFuzzer timed out								
Scanners_findUrlEnd	nibor.autolink.internal.Scanners.findUrlEnd	111	Subject considered for Scalability Evaluation (RQ3)* — SpecFuzzer timed out								

* Subjects for which SpecFuzzer timed out during mutation analysis are considered for Scalability Evaluation (RQ3).

VII. EXPERIMENTAL RESULTS

A. Prediction Evaluation (RQ1)

Figure 4 shows a Venn diagram recording the distribution of *Assertion Inferring Mutants* and *subsuming* mutant sets. We can observe that the set of subsuming mutants is significantly different from the set of *Assertion Inferring Mutants*. Only a small number of subsuming mutants (75 out 264) are also assertion inferring, while a large number of *Assertion Inferring Mutants* (450 out of 525) are not subsuming. This shows that subsuming mutant selection is not well suited for the assertion inference task. Moreover, the set of assertion-inferring mutants represents 12.9% of the killable mutants, suggesting that an effective mutant selection strategy would allow for drastic assertion inference overhead reductions.

Venn diagram from Figure 5 shows that *Seeker* detects almost half of *Assertion Inferring Mutants* (258 out 525). Table I depicts Seeker's prediction performance across all 40 projects. Overall, *Seeker* predicts *Assertion Inferring Mutants* with 0.79 Precision, 0.49 Recall, and 0.58 MCC, a much better performance than random mutant selection (whose MCC value is 0). Thus, *Seeker* can provide significant improvements in terms of inferred assertions over baseline methods.

Answer to RQ1: *Seeker* predicts *Assertion Inferring Mutants* with 0.58 MCC, 0.79 Precision, and 0.49 Recall. The class of subsuming mutants cannot reliably select *Assertion Inferring Mutants* (only 28% of the subsuming mutants are also assertion inferring).

B. Inference Evaluation (RQ2)

Table II records SpecFuzzer's performance w.r.t. assertion inference by employing different mutant sets, i.e, *Subsuming Mutant Selection*, *Seeker*, and *Random Mutant Selection*. The results show that when SpecFuzzer uses *Seeker*'s predicted mutants, it infers 87.51% of total assertions, i.e., only 12.49% of the assertions ares missed (the cost of considering only *Seeker*'s predicted mutants) with 46.29 times faster mutation analysis than using all the mutants (and 2.5 times faster than considering subsuming mutants). *Seeker* enables SpecFuzzer to infer at least one assertion for all subjects, and successfully infers all assertions for 23 subjects.

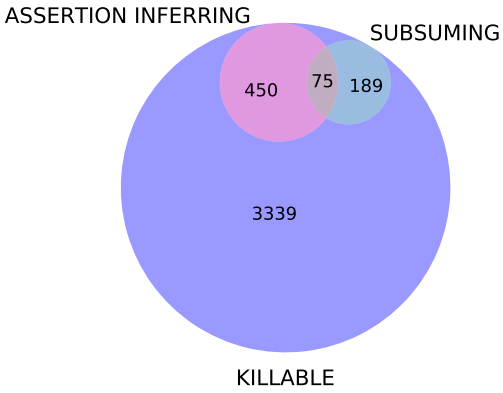

Fig. 4. Mutant Class distribution

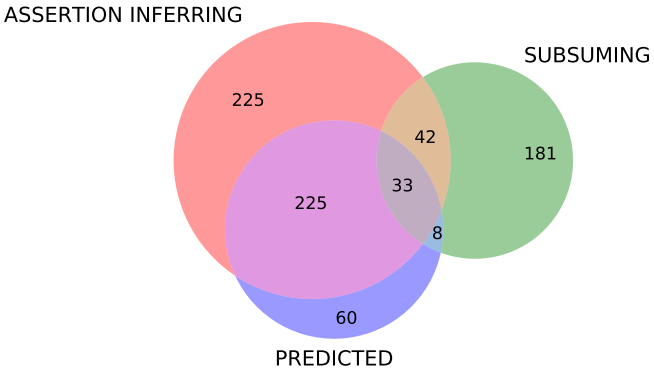

Fig. 5. Prediction distribution

When SpecFuzzer uses the subsuming mutants, it infers 57.77% of total assertions. It infers all assertions for 5 subjects but fails to infer any for 7 subjects. Although it misses 42.23% of the assertions (the cost of considering only subsuming mutants), diminishing the benefit of an improved mutant analysis time (19.16 times faster than using all mutants). A good improvement in the mutation testing time is noted when SpecFuzzer uses randomly selected mutants, but it fails to infer 48.53% of total assertions. In 2 cases it infers all assertions and fails to infer any assertion for 2 other cases. Overall, *Seeker* outperforms both, random and subsuming mutant selection, with a statistically significant[3] sizeable difference.

Answer to RQ2: *Seeker* enables SpecFuzzer to infer assertions for all subjects, running 46.29 times faster at the expense of 12.49% of the assertions. At the same time, *Seeker* enables SpecFuzzer to infer 36% and 30% more assertions than *Random Mutant Selection* and *Subsuming Mutant Selection*, runs 2.5 times faster than *Subsuming Mutant Selection* and requires similar execution time (wall clock) to *Random Mutant Selection*.

C. Scalability Evaluation (RQ3)

Table III records the results of SpecFuzzer's performance in inferring assertions when it employs *Seeker* and *Random Mutant Selection*, for the subjects where mutation analysis with all mutants timed out. *Seeker* selected 2.99% mutants from the entire mutant set. Among the predicted mutants, 83.33% mutants are assertion inferring. When an equal number of mutants are selected using *Random Mutant Selection*, only 16.67% of mutants selected are assertion inferring. When SpecFuzzer considers only *Seeker*'s predicted mutants for assertion filtering, it infers assertions for all subjects mentioned in Table III within 16 minutes, on average. On the other hand, for 50% of the subjects (3 out of 6), SpecFuzzer fails to infer any assertion if it uses *Random Mutant Selection*.

[3]We compared the inferred assertion percentages using Wilcoxon sign-rank-test and obtained a $p - value < 0.05$.

TABLE II
RQ2 RESULTS - PERFORMANCE OF ASSERTION INFERENCE

Mutation filtered assertion inference			
	With Subsuming Mutant Selection	**With *Seeker***	**With Random Mutant Selection**
Inferred Assertions (per Subject)	57.77%	87.51%	51.47%
Missed Assertions (Cost)	42.23%	12.49%	48.53%
Improvement in Time (Benefit)	19.16 times	46.29 times	47.34 times
Subjects with assertions inferred			
Total Subjects# 40	**With Subsuming Mutant Selection**	**With *Seeker***	**With Random Mutant Selection**
Subjects with All assertions inferred	5	23	2
Subjects with No assertion inferred	7	0	2

TABLE III
RQ3 RESULTS - SCALABILITY EVALUATION

***Assertion Inferring Mutants* (among mutants selected)**		
Mutants selected: 2.99% from the entire mutant set (per subject)	**With *Seeker***	**With Random Mutant Selection**
Assertion Inferring Mutants (among selected mutants)	83.33%	16.67%
Inferred assertions#		
Subject	**With *Seeker***	**With Random Mutant Selection**
EmailScanner_findFirst	85	58
EmailScanner_scan	192	0
IdentityHashSet_isEmpty	3	2
OptionGroup_setRequired	8	8
OptionGroup_setSelected	8	0
Scanners_findUrlEnd	23	0

Answer to RQ3: *Seeker* enables SpecFuzzer to scale by inferring assertions for all subjects where a complete mutation analysis timed out and *Random Mutant Selection* failed in 50% of the cases.

VIII. DISCUSSION

In the work of *Molina et al.* [37], the authors carefully studied the subjects and manually produced corresponding *Ground Truth* assertions capturing the intended behavior of the subjects. SpecFuzzer [37] was able to infer the ground truth assertions for 26 subjects, when all mutants were considered for assertion inference. Hence, we also compared the effectiveness of all three mutant selection techniques in inferring *Ground Truth* assertions and assessed how *Seeker* compares with the subsuming and randomly selected mutants in terms of inferred ground truth assertions.

TABLE IV
DISCUSSION - INFERRING GROUND TRUTH ASSERTIONS

Ground Truth assertion inference			
	With Subsuming Mutant Selection	**With *Seeker***	**With Random Mutant Selection**
Inferred Assertions (per Subject)	67.31%	96.15%	19.23%
Subjects with assertions inferred			
Total Subjects# 26	**With Subsuming Mutant Selection**	**With *Seeker***	**With Random Mutant Selection**
Subjects with All assertions inferred	17	25	5
Subjects with No assertion inferred	8	1	21

Table IV records SpecFuzzer's performance in ground truth assertion inference by employing the different mutant selection techniques. On considering *Seeker*'s predicted mutants, SpecFuzzer infers almost all (96.15%) ground truth assertions, which is superior to both *Random Mutant Selection* (infers 19.23%) and *Subsuming Mutant Selection* (infers 67.31%). Also, *Seeker*'s predicted mutants enable SpecFuzzer to infer at least one ground truth assertion for all subjects except for one subject (doublylinkedlistnode_insertRight). This evidences that *Seeker* can help in inferring quality assertions, in this case, human-written and manually validated assertions.

IX. THREATS TO VALIDITY

External Validity: Threats may relate to the subjects we used. Although our evaluation expands to projects of various sizes, the results may not generalize to other projects. We consider this threat of low importance since we have a large sample of subjects (40 subjects from the previous studies [37], [38], [50] and 6 subjects from GitHub for scalability evaluation). Moreover, our predictions are based on the local mutant context, that has been shown to be determinant of mutants' utility [23], [27]. Other threats may relate to the assertion inference technique that we used for evaluation. This choice was made since SpecFuzzer is the current state of the art and operates similarly to other techniques (the main differences lie in the grammar used). We consider this threat of low importance since *Seeker* deals with mutation analysis, which is used in the same way by all assertion inference techniques [37], [38], [50], and are directly impacted by the number of mutants involved. Nevertheless, in case other techniques require different predictions, one could re-train, tune and use *Seeker* for the specific method of interest, as we did here with SpecFuzzer.

Internal Validity: Threats may relate to the restriction that we impose on sequence length, i.e., a maximum of *500* tokens. This was done to enable reasonable model training time, approximately *24* hours to learn mutant embeddings on a Tesla V100 gpu. Other threats may be due to the use of *tf-seq2seq* [1] for learning mutant embeddings. This choice was made for simplicity, to use the related framework out of the box, similar to related studies [21], [22], [52]. Other internal validity threats could be related to the test suites we used and the mutants considered as assertion inferring. To deal with this issue, we used well-tested programs and state-of-the-art tools to generate extensive pools of tests (Evosuite [18] and Randoop [40]) as done by previous work [37], [38], [50]. This is also a typical process followed in mutation testing studies [23], [27], [32], [41]. To be more accurate, our underlying assumption is that the extensive pool of tests used in our experiments is a reasonable approximation of the program's test executions.

Construct Validity: Our assessment metrics (assertions inferred and incurred time during mutation analysis) may not reflect the actual cost or benefit values. These metrics are intuitive, i.e., the inferred assertions are the output of assertion inference techniques, and the incurred time during mutation analysis is the wall clock time these techniques invest in filtering assertions. We mitigate these threats by following suggestions from mutation testing and assertion inference literature, using state-of-the-art tools, performing multiple simulations, and confirming consistent results across subjects.

X. CONCLUSION

We introduced the notion of *Assertion Inferring Mutants* and demonstrated that this small subset of mutants (i.e., 12.95% mutants of the entire mutant set) is sufficient to effectively identify strong assertions. We also showed that this mutant set is significantly different from *Subsuming Mutants* (i.e., 71.59% of assertion inferring mutants are not subsuming). Though subsuming mutants have been frequently cited in the mutation testing literature and have been shown to improve efficiency, they are not sufficient for the assertion inference task as their use results in losing almost half of the strong assertions. We also explored a learning-based approach, in particular, the widely adopted encoder-decoder architecture that can learn to statically identify assertion inferring mutants from the given mutant sets, given their contextual information. Our experiments on 40 subjects show that it identified assertion inferring mutants with 0.58 MCC, 0.79 Precision, and 0.49 Recall. These predictions enabled 42.29 times faster assertion inference with minor effectiveness loss (12.49% fewer assertions inferred) compared to the use of all mutants. Moreover, it enabled assertion inference technique SpecFuzzer to scale on all our considered large subjects (by inferring assertions where SpecFuzzer timed out previously) in comparison to *Random Mutant Selection* which failed to infer any assertion in 50% of the cases.

XI. DATA AVAILABILITY

In addition to our dataset consisting of all projects' source code, augmented test suites, generated mutants, and SpecFuzzer execution logs, *Seeker*'s source code with additional plots is also publicly available in our GitHub repository[4].

ACKNOWLEDGMENT

This work is supported by the Luxembourg National Research Fund (FNR) through the CORE project under Grant C19/IS/13646587/RASoRS.

[4]https://github.com/garghub/seeker

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[3] Paul Ammann, Márcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 21–30. IEEE Computer Society, 2014.

[4] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[6] Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. End-to-end attention-based large vocabulary speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2016, Shanghai, China, March 20-25, 2016*, pages 4945–4949. IEEE, 2016.

[7] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Trans. Software Eng.*, 41(5):507–525, 2015.

[8] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 242–253. ACM, 2018.

[9] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

[10] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc V. Le. Massive exploration of neural machine translation architectures. *CoRR*, abs/1703.03906, 2017.

[11] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, and Koushik Sen. Selecting fault revealing mutants. *Empir. Softw. Eng.*, 25(1):434–487, 2020.

[12] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, and Koushik Sen. Selecting fault revealing mutants. *Empir. Softw. Eng.*, 25(1):434–487, 2020.

[13] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734. ACL, 2014.

[14] Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Softw. Eng. Notes*, 31(3):25–37, 2006.

[15] Marcelo d'Amorim, Carlos Pacheco, Tao Xie, Darko Marinov, and Michael D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pages 59–68. IEEE Computer Society, 2006.

[16] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. Inference and enforcement of data structure consistency specifications. In Lori L. Pollock and Mauro Pezzè, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 233–244. ACM, 2006.

[17] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[18] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 416–419. ACM, 2011.

[19] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Trans. Software Eng.*, 38(2):278–292, 2012.

[20] Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias. Analysis of invariants for efficient bounded verification. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 25–36. ACM, 2010.

[21] Aayush Garg, Renzo Degiovanni, Matthieu Jimenez, Maxime Cordy, Mike Papadakis, and Yves Le Traon. Learning from what we know: How to perform vulnerability prediction using noisy historical data. *Empir. Softw. Eng.*, 27(7):169, 2022.

[22] Aayush Garg, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Vulnerability mimicking mutants. *CoRR*, abs/2303.04247, 2023.

[23] Aayush Garg, Milos Ojdanic, Renzo Degiovanni, Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. Cerebro: Static subsuming mutant selection. *IEEE Transactions on Software Engineering*, pages 1–1, 2022.

[24] Dunwei Gong, Gongjie Zhang, Xiangjuan Yao, and Fanlin Meng. Mutant reduction based on dominance relation for weak mutation testing. *Information & Software Technology*, 81:82–96, 2017.

[25] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. On the limits of mutation reduction strategies. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 511–522. ACM, 2016.

[26] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 695–705, New York, NY, USA, 2019. Association for Computing Machinery.

[27] René Just, Bob Kurtz, and Paul Ammann. Inferring mutant utility from program context. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 284–294, 2017.

[28] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: an efficient and extensible tool for mutation analysis in a java compiler. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 612–615. IEEE Computer Society, 2011.

[29] Nal Kalchbrenner and Phil Blunsom. Recurrent continuous translation models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP 2013, 18-21 October 2013, Grand Hyatt Seattle, Seattle, Washington, USA, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1700–1709. ACL, 2013.

[30] Marinos Kintis, Mike Papadakis, and Nicos Malevris. Evaluating mutation testing alternatives: A collateral experiment. In Jun Han and Tran Dan Thu, editors, *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*, pages 300–309. IEEE Computer Society, 2010.

[31] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio E. Delamaro, Mariet Kurtz, and Nida Gökçe. Analyzing the validity of selective mutation with dominator mutants. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 571–582, New York, NY, USA, 2016. Association for Computing Machinery.

[32] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio Eduardo Delamaro, Mariet Kurtz, and Nida Gökçe. Analyzing the validity of selective mutation with dominator mutants. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 571–582. ACM, 2016.

[33] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, 2005.

[34] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 133–146. ACM, 2012.

[35] Michaël Marcozzi, Sébastien Bardin, Nikolai Kosmatov, Mike Papadakis, Virgile Prevosto, and Loïc Correnson. Time to clean your test objectives. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 456–467. ACM, 2018.

[36] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[37] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. Fuzzing class specifications. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1008–1020. ACM, 2022.

[38] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. Evospex: An evolutionary algorithm for learning postconditions. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1223–1235. IEEE, 2021.

[39] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, 1996.

[40] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 75–84. IEEE Computer Society, 2007.

[41] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 354–365. ACM, 2016.

[42] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: An analysis and survey. *Adv. Comput.*, 112:275–378, 2019.

[43] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: An analysis and survey. *Adv. Comput.*, 112:275–378, 2019.

[44] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Gregory T. Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. Automatically patching errors in deployed software. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 87–102. ACM, 2009.

[45] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. What is the vocabulary of flaky tests? In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 492–502, New York, NY, USA, 2020. Association for Computing Machinery.

[46] J. K. Rowling. *Harry Potter and the Philosopher's Stone*, volume 1. Bloomsbury Publishing, London, 1 edition, June 1997.

[47] Martin J. Shepperd, David Bowes, and Tracy Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Trans. Software Eng.*, 40(6):603–616, 2014.

[48] Apeksha Shewalkar, Deepika Nyavanandi, and Simone A. Ludwig. Performance evaluation of deep neural networks applied to speech recognition: Rnn, LSTM and GRU. *J. Artif. Intell. Soft Comput. Res.*, 9(4):235–245, 2019.

[49] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112, 2014.

[50] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. Evolutionary improvement of assertion oracles. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1178–1189. ACM, 2020.

[51] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

[52] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 25–36. IEEE / ACM, 2019.

[53] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4):19:1–19:29, 2019.

[54] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*, pages 301–312. IEEE, 2019.

[55] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1398–1409. ACM, 2020.

[56] Jingxiu Yao and Martin J. Shepperd. Assessing software defection prediction performance: why using the matthews correlation coefficient matters. In Jingyue Li, Letizia Jaccheri, Torgeir Dingsøyr, and Ruzanna Chitchyan, editors, *EASE '20: Evaluation and Assessment in Software Engineering, Trondheim, Norway, April 15-17, 2020*, pages 120–129. ACM, 2020.

[57] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. Operator-based and random mutant selection: Better together. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 92–102. IEEE, 2013.

[58] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. Is operator-based mutant selection superior to random mutant selection? In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 435–444. ACM, 2010.