MoFuzz: A Fuzzer Suite for Testing Model-Driven Software Engineering Tools

Hoang Lam Nguyen
nguyehoa@informatik.hu-berlin.de
Humboldt-Universität zu Berlin
Germany

Nebras Nassar
nassarn@informatik.uni-marburg.de
Philipps-Universität Marburg
Germany

Timo Kehrer
kehrer@informatik.hu-berlin.de
Humboldt-Universität zu Berlin
Germany

Lars Grunske
grunske@informatik.hu-berlin.de
Humboldt-Universität zu Berlin
Germany

ABSTRACT

Fuzzing or fuzz testing is an established technique that aims to discover unexpected program behavior (e.g., bugs, security vulnerabilities, or crashes) by feeding automatically generated data into a program under test. However, the application of fuzzing to test Model-Driven Software Engineering (MDSE) tools is still limited because of the difficulty of existing fuzzers to provide structured, well-typed inputs, namely models that conform to typing and consistency constraints induced by a given meta-model and underlying modeling framework. By drawing from recent advances on both fuzz testing and automated model generation, we present three different approaches for fuzzing MDSE tools: A graph grammar-based fuzzer and two variants of a coverage-guided mutation-based fuzzer working with different sets of model mutation operators. Our evaluation on a set of real-world MDSE tools shows that our approaches can outperform both standard fuzzers and model generators w.r.t. their fuzzing capabilities. Moreover, we found that each of our approaches comes with its own strengths and weaknesses in terms of fault finding capabilities and the ability to cover different aspects of the system under test. Thus the approaches complement each other, forming a fuzzer suite for testing MDSE tools.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; **Software testing and debugging**.

KEYWORDS

Model-Driven Software Engineering, Modeling Tools, Fuzzing, Automated Model Generation, Eclipse Modeling Framework

ACM Reference Format:
Hoang Lam Nguyen, Nebras Nassar, Timo Kehrer, and Lars Grunske. 2020. MoFuzz: A Fuzzer Suite for Testing Model-Driven Software Engineering Tools. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416668

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416668

1 INTRODUCTION

Fuzz testing (aka. fuzzing) automatically generates a large number of inputs and feeds them to the program under test to discover unexpected program behavior and evaluate the program's reliability. Since its initial introduction by Miller et al. [48], fuzzing via tools like AFL [93, 94] has been widely adopted due to its conceptual simplicity and its proven effectiveness [29, 44, 95]. Various advanced approaches have been proposed to improve the performance of fuzzers through guiding the input generation process [9, 10, 43, 45]. Further improvements have been made by including targeted input string mutations [63] or problem-specific generators [61].

However, the application of fuzzing to test Model-Driven Software Engineering (MDSE) tools is still limited. This is due to the difficulty of existing fuzzers to provide structured, well-typed inputs, namely models that conform to typing and consistency constraints induced by a given meta-model and underlying modeling framework.

In this paper, we investigate the fuzzing of MDSE tools which are based on the Eclipse Modeling Framework (EMF) [74], a reference implementation of OMG's Essential Meta-Object Facility (EMOF) [59], which has emerged as a de-facto standard in the context of open source MDSE environments and tool chains. Tools working with EMF models as input have a processing pipeline as illustrated in Figure 1. The parsing stage translates a model from some external representation (typically an XMI file [58]) into an internal representation (an object structure which can be conceptually considered as an abstract syntax graph (ASG)) which can be easily processed in terms of the core functionality (e.g., code generation, model editing, model comparison, etc.) of the tool. Models passing the parsing stage are syntactically valid in the sense that they respect basic consistency constraints imposed by EMF (aka. valid EMF models [7, 50]) and they are properly typed w.r.t. the given meta-model. Optionally, syntactically valid models may need to pass a validation stage checking further semantic consistency constraints, notably multiplicity constraints or arbitrary well-formedness rules (e.g., formulated in OCL [57]) defined by the meta-model. Parsing and validation correspond to what is referred to as syntactic and semantic stage in [61], respectively. Input models may be rejected

by each of these stages, and the tool's core functionality remains untested. Consequently, fuzzers for effectively testing EMF-based tools must generate input models which at least adhere to EMF and typing constraints, a requirement which is not met by any of the existing fuzzing approaches.

We present MoFuzz, a fuzzer suite for testing MDSE tools which tackles this challenge by drawing from two research fields that have developed independently of each other, namely (i) mutation- and generator-based fuzzing (see Section 2.1), and (ii) automated model generation (see Section 2.2). Basically, the former provides inspiration to guide a fuzzer towards effectively covering the system under test (i.e., in our case, the MDSE tool under test), while the latter provides the technology to generate valid models. *While this idea appears to be obvious, it is yet unclear which combinations of fuzzing approaches and technologies for model generation are the most effective ones, which constitutes the main research question we aim to answer in this paper.*

We propose three different fuzzers to generate models that serve as inputs for fuzzing MDSE tools: (i) a graph grammar-based fuzzer that attempts to cover different aspects of the input domain as fast as possible, and two mutation-based approaches that aim to incrementally evolve inputs to exercise deep paths using either (ii) a set of customly defined mutations built on top of the EMF Edit API, or (iii) a set of edit operations specified as model transformation rules which are generated from the given meta-model [40]. The graph grammar-based fuzzer is inspired by the idea of using a constructive specification of the input format to generate syntactically valid inputs by construction. While this idea has been explored for textual input formats using context-free grammars, our approach is integrated with an EMF Model Generator [50] which is based on the concept of instance-generating graph grammars. Our mutation-based fuzzers are guided in the sense that promising seed inputs are selected by a feedback loop that leverages coverage information from the system under test. Their variation point lies in the level of consistency being preserved by the model mutations.

We have evaluated our fuzzing approaches on a set of real-world MDSE tools based on EMF. Our experimental results show that all approaches can outperform existing fuzzers and automated model generators w.r.t. the effectiveness of their fuzzing capabilites. Moreover, we found that each of our approaches comes with its own strengths and weaknesses in terms of fault finding capabilities and the ability to cover different aspects of the system under test. Thus the approaches complement each other, forming a fuzzer suite for testing MDSE tools which can contribute to increase the maturity of modeling tools. This is still considered as one of the biggest obstacles for adopting MDSE in practice [85].

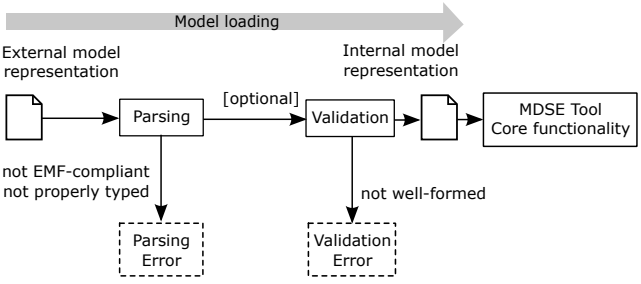


Figure 1: Processing pipeline of EMF-based MDSE tools: Models need to be loaded into an ASG-based internal representation to be processed by the tool; they must be at least EMF-compliant and properly typed w.r.t. the given meta-model such that the tools's core functionality is reached.

2 STATE-OF-THE-ART AND RELATED WORK

Our work on MoFuzz builds on existing approaches in two areas, namely work on fuzzing in general and work on automated model generation. In the following, we review the state-of-the-art on both areas and highlight the need for our research on MoFuzz.

2.1 Fuzzing

The main idea behind fuzzing [48] is to test the program by repeatedly feeding it with random input data that may be syntactically or semantically malformed. Usually, fuzzing approaches are classified based on the level of program analysis that is employed: *blackbox*, *whitebox*, or *greybox* fuzzing [29, 44]. Besides, fuzzing tools can also be classified based on how they produce new test inputs into *mutation-based* and *generator-based* fuzzers.

2.1.1 Mutation-based Fuzzing. Mutation-based Fuzzers like zzuf [14] or AFL [94] generate new inputs by randomly mutating well-formed seed inputs. The mutations usually operate on raw test inputs represented as a sequence of bytes, and ignore complex input (file) structures.

Advances in fuzzing [9, 10, 43, 45] mostly operate on providing guidance for the mutations and the search process to improve the coverage of the system under test. Specifically, tools like AFLFast [10] use a coverage-guided greybox fuzzing approach to systematically explore the input state space and prioritize mutations of test inputs in the fuzzing corpus that cover novel program areas. Based on this idea, AFLGo [9] extended the coverage-guided fuzzing idea towards *directed greybox fuzzing*, by steering the search through selecting inputs that help to reach specific code areas instead of novel ones. In contrast, FairFuzz [43] and Steelix [45] do not determine which input to mutate, but how to mutate the input. FairFuzz proposes a targeted mutation strategy that tries to focus the location in the input string that will likely generate mutants to hit rarely covered branches. Similarly, Steelix uses light-weight static analysis to gain information that pinpoints interesting bytes in the input.

The study by Pham et al. [63] shows that randomness of byte mutations can be ineffective and hence they use dedicated string-based mutation operators that correspond to the input structure in their tool AFLSmart. They showed that AFLSmart can improve coverage and identify more vulnerabilities than AFLFast and AFL.

2.1.2 Generator-based Fuzzing. Generator-based fuzzing addresses the limitation of traditional fuzzers like AFL, which are likely to fail to generate valid inputs for programs which expect highly structured inputs [34, 37, 83].

To tackle this problem, generator-based fuzzers such as Peach [62] or Csmith [88] leverage domain-specific knowledge of the input format to generate input files that are *syntactically valid* by construction. The generated input files are thus more likely to pass the early input processing stages and to test the core functionality of

the program. Some tools use file format specifications [2, 62] or a context-free grammar [34, 37, 80] to describe the syntax of inputs.

Generator-based fuzzers do not require seed inputs. Instead, the input files are often generated *at random* using the file format specification. A drawback of these approaches is that while the generated inputs may be syntactically valid, they are likely still *semantically invalid*. For instance, a syntactically correct JavaScript file that refers to variables that are not defined in the current execution context will result in an error in the semantic analyzer.

2.1.3 Hybrid Fuzzing Approaches. There are several approaches that try to combine mutation-based fuzzing methods with generator-based methods in order to mitigate their weaknesses.

For example, Skyfire [82] learns a probabilistic context-sensitive grammar from an initial corpus and a context-free grammar in order to produce a corpus that consists of valid and well-distributed inputs. The generated corpus can then be fed to any mutation-based fuzzer (e.g., AFL) to improve the code coverage, thereby increasing its likelihood of finding vulnerabilities.

In contrast, LangFuzz [37] and jsfunfuzz [67] use a tight integration of generator- and mutation-based fuzzing in order to generate syntactically correct JavaScript code fragments. Superion [83] extends Skyfire and applies a greybox fuzzing technique which uses grammar-aware trimming and mutation strategies. The evaluation shows that this approach improves the bug finding capabilities and code coverage compared to AFL and jsfunfuzz [67].

Finally, Zest [61], integrates ideas of coverage-guided fuzzing (i.e. mutation-based fuzzing with coverage-feedback) and random generator-based fuzzing. In particular, Zest transforms a (pseudo) random input generator into a deterministic *parametric input generator* by mapping the random choices of the generator to fixed bit-parameter sequences. These parameter sequences can be mutated using common bit-level mutation operators in order to directly control the input generation process. By leveraging code coverage and validity feedback, Zest can then automatically guide the parametric generator to produce inputs that effectively test the semantic analysis stage.

2.2 Automated Model Generation

In the last decade, the automatic generation of instances of a given meta-model has gained considerable attention by the MDSE research community. In the sequel, we review existing approaches w.r.t. to their capability to generate valid EMF models, which has emerged as a de-facto standard in MDSE. The internal structure of an EMF model can be conceptually considered as typed attributed graph (aka. ASG) whose node and edge types are drawn from a meta-model serving as type graph; a formal treatment of typing conformance can be found, e.g., in [7]. In the sequel, we stick to the object-oriented terminology which is more common in the EMF community, i.e., nodes and edges of a model's ASG are referred to as objects and references, respectively, where containments are a specific kind of reference. An EMF model is said to be valid if it adheres to all EMF constraints as formalized in [7]. Most notably, each object must not have more than one container and cycles of containments must not occur. Semantic validity is specified by additional well-formedness constraints defined by a meta-model, such as multiplicity constraints or arbitrary OCL expressions.

Existing approaches to automated model generation vary substantially w.r.t. the level of consistency of the generated models and the performance of the generation algorithm. Typically, the higher the consistency guarantees, the slower and less scalable the model generation. From a technical point of view, existing model generation approaches can be classified as follows.

2.2.1 Solver-based Approaches. Solver-based approaches generate models by translating a meta-model into a logical formula and using an off-the-shelf solver to find possible solutions. The approaches presented in [46, 47, 70, 78] use Alloy [38] for this purpose. Other approaches are based on CSP modeling [30, 35] or rely on SMT solvers [87]. While solver-based approaches are capable of generating models that respect arbitrary well-formedness constraints, their scalability is limited to generating pretty small models [69]. More recently, a graph solver known as VIATRA Solver [68, 69] has been presented for the automated generation of valid EMF models respecting arbitrary well-formedness rules. While experimental results indicate that the approach performs better than existing approaches using Alloy, its scalability and efficiency is still limited [68, 69].

2.2.2 Grammar-based Approaches. Grammar-based approaches translate a given meta-model serving as declarative language specification into a constructive one which is then used for the sake of model generation. Mougenot et al. [49] convert a given meta-model into a tree grammar which is used to uniformly create tree structures. Uniformity means that for a tree of given size (in number of nodes), the method is capable of uniformly generating all tree structures of that size. However, models are reduced to their containment structure, which is a severe oversimplification in practice.

Ehrig et al. [27] present an approach for converting type graphs with restricted multiplicity constraints into instance-generating graph grammars that can be used to systematically enumerate all properly typed models of a modeling language. Taentzer [79] generalizes that approach to arbitrary multiplicies. Recently, Nassar et al. [50] developed the EMF Model Generator, which extends and advances the ones in [27, 79] and respects EMF constraints as well.

2.2.3 Mutation-based Approaches. Mutation-based generators take a (potentially empty) model as input and manipulate it by applying model modifications. The SiDiff model generator (SMG) proposed by Pietsch et al. [64] performs model manipulation by applying model editing operations. However, the SMG needs to be configured by a stochastic controller which needs to be trained on a real-world model history. It has mainly been developed as a testbed for evaluating different learning strategies [89–92]. In contrast, the AtlanMod EMF Random Instantiator [5], an EMF-based tool which is still under active devlopment, can be used out of the box. It generates EMF models by using probability distributions for each meta-class and property defined by the meta-model, with uniform distributions as the default configuration.

2.3 Need for Further Research

Mutation-based and generator-based fuzzers are applied to create test inputs in different input-file formats, including XSLT and XML [9, 43, 61, 82, 83], jpeg and png [9, 43], Javascript [61, 83], just to name a few. However, while some of them, notably XML formats,

are similar to external representations of models serialized in XMI, there is a fundamental difference w.r.t. to their internal structure. First, structured representations of models form an *abstract syntax graph* (ASG), instead of a *strictly hierarchical* document object model (aka. DOM tree). Second, this complexity is further aggravated by the fact that a model's ASG must at least be properly typed and adhere to validity constraints of the underlying modeling framework (e.g., EMF) to pass the model loading process (cf. Figure 1). Such consistency constraints in the input files are typically not enforced by traditional XML-based tools (e.g., office environments or web browsers) which have been considered as system under test in previous approaches on structure-aware fuzzing.

On the contrary, existing work on automated model generation provides the technology to generate syntactically or even semantically valid models. However, there has been limited research on their capabilities for the sake of fuzz testing MDSE tools. Previous applications of automated model generation include benchmarking model queries and transformations [6, 77], model-driven search and optimization [8, 96], or validating the suitability of MDSE tools to deal with large input models [1, 19, 33]. An application scenario which is close to ours is model transformation testing. This has been first addressed by Fleurey et al. [31], motivated by the concept of equivalence classes in black-box testing, and later been extended and implemented in [13, 42, 84]. The basic idea is that equivalence classes are formed by certain meta-model patterns, and the generated models should represent suitable representatives of these equivalence classes. The goal, however, is to find bugs in application-specific model transformations, while we aim for testing generic or language-specific MDSE tools.

The closest related work focuses on the automated differential testing of cyber-physical system (CPS) tool chains (e.g., Matlab/Simulink). CyFuzz [16] randomly generates (possibly invalid) models and attempts to iteratively fix them based on the error messages of the system under test. However, since this approach is unable to produce large, realistic models, it has been subsequently improved by incorporating semi-formal specifications [17] as well as CPS language-specific mutation techniques [18]. While these techniques have been shown to be effective, they are tightly coupled with Simulink, which limits the portability to other modeling languages and MDSE tool suites. In contrast, our goal is to develop a fuzzer suite which is based on EMF and thus applicable to a wider range of general-purpose as well as domain-specific MDSE tools. Another technique, DeepFuzzSL [72], learns a language model from existing valid Simulink models and is able to generate further valid models with high probability, though it requires a large number of seed models for deep learning. Since their accessibility is often limited (see, e.g., [11]), we aim to abstain from relying on seed models but only require to have access to a meta-model that describes the abstract syntax as well as the static semantics of its instance models. This is a reasonable assumption since meta-models are the standard way to define a modeling language, and they can be expected to be publicly available in the context of the EMF ecosystem.

In sum, it is yet unclear which combinations of fuzzing approaches and technologies for model generation are the most effective ones for fuzz testing MDSE tools, which constitutes the guiding research question we aim to answer in this paper.

3 APPROACH

3.1 Overview

A general overview of our fuzzer suite called MoFuzz is given in Figure 2. It takes as input the meta-model of the input domain as well as the MDSE tool to be tested. Internally, we build upon JQF [60] (version 1.4), a feedback-directed fuzz testing framework for Java. JQF performs on-the-fly bytecode instrumentation to measure code coverage and repeatedly executes the program under test using inputs provided by custom input generators. Furthermore, JQF provides implementations for various fuzzing algorithms, including the Zest algorithm (see Section 2.1.3), which will be used as a baseline technique in our evaluation.

We provide three different input generation strategies: (i) a graph grammar-based method that attempts to cover different aspects of the input domain as fast as possible by leveraging a recently introduced efficient model generator [50], and two mutation-based approaches that aim to incrementally evolve inputs to exercise deep paths using either (ii) a set of customly defined mutations built on top of the generic EMF Edit API, or (iii) a set of consistency-preserving edit operations (CPEOs) specified as model transformation rules which are generated from the given meta-model using the approach [40] and supporting tool suite presented in [39, 66].

3.2 Generator-Based Fuzzing

The main aim of using a generator-based fuzzing approach is to generate models covering different aspects of the input domain as fast as possible while adhering to a high level of consistency. Although the strongest consistency guarantees are provided by solver-based generators (see Section 2.2), existing tools are way too inefficient for our purpose of fuzzing; they fail to generate large numbers of models in a short amount of time. From the remaining model generators, the EMF Model Generator which has been recently presented by Nassar et al. [50] represents the most promising trade-off between consistency and performance. The generated models are valid EMF models which are properly typed and which respect well-formedness rules up to arbitrary multiplicity constraints defined by the meta-model. Additionally, the tool can generate large valid models (with up to half a million elements) in a few minutes (in less than 2 minutes for simple meta-models and in about 30 minutes for more comprehensive ones). For the sake of self-containedness, we briefly recall the general model generation process, before we present the specific configuration which is integrated into our fuzzing suite MoFuzz.

3.2.1 Graph Grammar-based Model Generation. Following the typical schema of grammar-based approaches to model generation, the EMF Model Generator translates a meta-model into a constructive language specification (i.e., the grammar) which can be then used for the sake of model generation. Here, the grammar is realized as a model transformation system (MTS) which is implemented in the rule-based transformation language Henshin [4, 76]. Different kinds of rules and transformation units are designed to increase the model size, handle and improve the degree of violations, and to support user parameterizations. The MTS can be then used to generate EMF models according to various strategies. Typically, the model generation is performed in two phases. First, the *model*

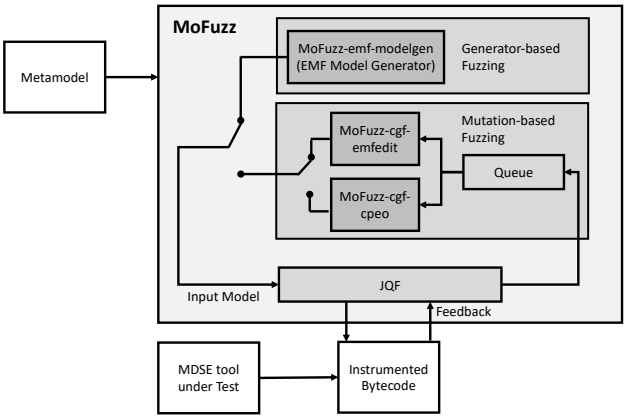


Figure 2: High-level overview of MoFuzz.

increase phase creates model elements without violating upper multiplicity bounds. Then, the *model completion* phase completes the intermediate model to a valid EMF model through interacting with the model repair tool EMF Repair [51, 52].

3.2.2 MoFuzz-Specific Generation Strategy. For the sake of fuzzing, we configure the model increase phase with two kinds of production rules: *additional-node-creation* and *additional-edge-creation*. The former create an ASG object of a certain type together with its incoming containment reference from a suitable container, and the latter inserts a cross-tree reference between two suitable existing objects. The rules are derived for each meta-class and reference type of the meta-model and equipped with application conditions to prevent introducing upper bound violations and to preserve the EMF constraints. All rules are wrapped into one so-called independent unit, which iteratively executes one of its comprised rules in a non-deterministic way.

Derived from our overall aims of using a generator-based fuzzer, the intuitions behind this strategy are as follows: The rules are designed to be atomic, i.e., they perform basic changes, and thus all possible structures of a model can be produced. By construction, the rules cover all the meta-model types. Thus, they can *broadly explore* and produce all possible valid EMF models for the given meta-model. Moreover, finding a rule match is highly *efficient* since it requires occurrences of a small-sized graph pattern to be found. The classification into two different kinds of rules increases the possibility of finding an applicable rule. Additionally, their atomic design increases the *flexibility* of combining them in any order. Finally, this strategy starts the model generation process *from scratch*, and thus, it is independent of the quality of a given seed model which, in principle, could be fed as an input into the EMF Model Generator. This strategy can generate large models efficiently. In MoFuzz, however, we choose a setting that limits the model size to avoid scalability problems of executing the MDSE tools under test (the limit has been set to 500 elements in our experiments presented in Section 4).

3.3 Mutation-Based Fuzzing

In this approach, we propose to adopt one of the most widely used techniques in automated fault detection, namely coverage-guided greybox fuzzing (CGF) [9, 10, 43, 45, 61], to the domain of MDSE. Based on a set of initial inputs, CGF iteratively applies various mutations to the inputs to produce new test inputs. The key idea is then to retain only those new inputs that increase coverage, thus effectively exploring the program space in an iterative manner. In particular, we propose to implement CGF in two phases, a *generation phase* and a *mutation phase*:

In the *generation phase*, random seed models are generated until a pre-configured target meta-model coverage has been achieved. The goal is to have a diverse set of seed inputs that cover different aspects of the program under test. However, we initialize our input queue with only the most recent (e.g., 50 as used in our experiments) inputs that covered new parts of the MDSE tool under test. The intuition behind this approach is that the first inputs usually only cover program locations that can be trivially reached. In contrast, later inputs that are still able to execute additional coverage beyond those trivial paths may be more worthwhile to retain.

In the *mutation phase*, an input model is selected from the queue and a set of mutations is applied to it. As described earlier, if the mutated input executes new coverage, it is also added to the queue, otherwise it is discarded. This process is repeated until the timeout is reached or the user manually aborts the process. However, traditional fuzzers that utilize CGF usually operate on binary/textual inputs and therefore apply mutations on the bit-level, which is not very effective for the input domain of instance models. Thus, we require mutation operators that operate directly on the model-level instead.

We provide two concrete implementations of the above described two-phased approach, namely one that makes use of the generic EMF Edit API (Section 3.3.1) and one that is based on rule-based mutations (Section 3.3.2).

3.3.1 Custom Mutations Based on the Generic EMF Edit API. For the first technique, we use different kinds of model mutations in order to address the different requirements of each of the two phases. Since the mutation operators are implemented based on the EMF Edit API, we refer to this technique as MoFuzz-cgf-emfedit.

(1) Generation phase: For the the generation phase, we build upon the existing *EMF Random Instantiator* [5] developed by the NaoMod team, formerly known as AtlanMod (see Section 2.2.3). Similar to the approach by Mougenot et al. [49], the EMF random instantiator randomly generates EMF models by utilizing the containment hierarchy defined by a given meta-model in two steps: First, the containment tree is populated with randomly initialized objects in a depth-first manner until a pre-configured target size is reached. The tree traversal may be pruned at random points or if a pre-configured maximum depth is reached. The result is a model that conforms to the containment hierarchy defined by the meta-model. Second, cross-tree references are created based on the available types in the partial model. This is done by iterating through each object in the model and setting cross references for which a suitable opposite object exists.

To get a high diversity of instantiated model structures in the generated seed models, we adapt the original random selection strategy

by prioritizing uncovered classes and reference types defined by the meta-model. This is common for grammar-based fuzzers which operate on derivation trees, where previously uncovered expansion rules are prioritized to achieve higher grammar coverage.

(2) Mutation Phase: The mutations based on the EMF Edit API operate on individual objects in the model and thus do not require complex graph patterns to be found by a computationally intensive matching algorithm. In particular, the following mutations have been implemented:

- *Add object*: Adds one or more objects to the model according to the containment hierarchy of the meta-model. Meta-classes that have not been instantiated before are prioritized.
- *Delete object*: Deletes one object and all transitively contained objects, including all cross-tree references to other objects.
- *Change attributes*: Randomly mutates all attributes of an object.
- *Unset attributes*: Unsets all attributes of an object. That is, all attribute values are set to their default values.
- *Change cross references*: Changes the cross-tree references for all objects in the model.
- *Replace subtree*: Deletes a subtree in the containment tree of the model and replaces it with a randomly generated containment tree of a pre-configured size.

In principle, the mutation operations to be applied could be selected uniformly. However, the application of the *Delete object* mutation could lead to more drastic changes compared to the *Unset attributes* mutation. Further, the application of the *Change cross references* mutation is computationally very expensive and thus should be applied less frequently. Therefore, our default implementation prioritizes the mutation operators in the following order when selecting a random mutation to apply: (1) Addition of objects, (2) Change/Unset attributes and Replace subtree, (3) Change cross references, and (4) Delete object.

Compared to solver-based or graph-grammar-based methods, the EMF API based operations are more efficient as they do not require to perform any expensive constraint solving or graph transformation operations. However, only basic consistency constraints are considered, namely consistent containment hierarchy and proper typing; any further constraints (see Section 2.2) may be violated. We tolerate this since, with this technique, we favor speed over consistency. We rely on the evolutionary algorithm to filter out input models that are too malformed and thus do not expose new parts of the tested MDSE tool's core functionality.

3.3.2 Rule-Based Mutations Derived from the Meta-Model. Our second coverage-guided fuzzer, referred to as MoFuzz-cgf-cpeo in Figure 2, uses rule-based model transformations being derived from a given meta-model [40] as model mutations. The transformation rule generator has been originally designed with the aim of producing declarative specifications of edit operations (using the Henshin transformation language) which are available as editing commands in visual model editors. These edit operations must preserve the level of consistency being enforced by standard visual editors, which comprises basic typing and EMF constraints. In addition, a restricted set of multiplicities is enforced, notably lower bounds, in order to keep a model in a graphically displayable state. For example, associations in a UML class diagram must contain at least two association ends (aka. mandatory children), each of which must be connected to a suitable classifier (aka. mandatory neighbor). Compared to the custom mutations presented in the previous section, our main intuition of using such consistency-preserving edit operations (CPEOs) for fuzzing is to synthesize models on a higher level of consistency, which are thus supposed to more reliably pass the model loading pipeline depicted in Figure 1.

The general kinds of CPEOs are similar to our customly defined mutations, comprising operations for the creation, deletion, moving and changing of model elements:

- *Node creation* rules create an ASG object of a specific type. In contrast to our API-based object creations, the created object is immediately connected to its container, and mandatory children and neighbors are created, too. All attribute values of created objects are set along with their creation.
- *Edge creation* rules are limited to the creation of cross-tree references between existing source and target ASG objects.
- *Node and edge deletions* are performed by rules being inverse to node and edge creation rules, respectively.
- *Move and change* rules re-structure the relations between existing ASG objects. While a move rule moves an object from one container to another one, a change rule just changes the target object of a cross-tree reference. Likewise, attribute values can be changed from an old value to a new one.

Both the generation and the mutation phase of our fuzzer uses CPEOs from the above set. However, during the generation phase, only node and edge creation rules are selected in order to quickly reach a certain degree of meta-model coverage. During the mutation phase, all kinds of CPEOs are used to mutate the most promising models in our feedback loop queue.

In contrast to the strategy of applying API-based mutations, we choose a uniform probability distribution over all kinds of CPEOs for selecting the next mutation. The rationale behind this choice is that node deletions and movements cannot lead to drastic changes as it is the case for our API-based mutations. This is due to the fact that the edit rules specifying a CPEO are applied by the Henshin transformation engine by checking the so-called dangling condition imposed by the double-pushout approach to graph transformation [26]. The dangling condition prevents a rule from being applicable if it would lead to dangling references in the ASG, which in turns means that we cannot delete larger sub-graphs in a single transformation step specified by a CPEO. Larger sub-graphs can only be deleted in a stepwise manner, starting from the leafs of the containment tree.

4 EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation of our fuzzer suite; the source code of MoFuzz as well as the benchmark subjects are publicly available at: https://github.com/hub-se/MoFuzz.

4.1 Study Design

4.1.1 Experimental Subjects. The ideal form of evaluation would be to assess the performance of our fuzzer suite on an established benchmark like Google's FuzzBench [36], as is customary for traditional fuzzers and which is also required for sound experimental evaluation [41]. However, none of these benchmarks comprise

MDSE tools serving as program under test, and there is yet no benchmark for fuzz testing MDSE tools, which we are pioneering in this work. To that end, we selected a set of real-world MDSE tools as experimental subjects. All tools are publicly available, implemented in Java and on top of EMF and provide a suitable test driver serving as entry point for the fuzzing:

EMF2GraphViz [12] generates graphical representations of EMF models based on the GraphViz [28] utility. The test driver performs the generation using the default graphical description.

UMLValidator is a utility class of the Eclipse UML2 project [21] to validate all types of UML model elements. The test driver performs the validation of a given UML model by simply iterating through all model elements and validating them successively (this includes checking multiplicity constraints as well as verifying the presence of required containments and attributes).

UML2Java [53] generates Java source code from UML models. It is one of the examples delivered with the template-based code generator Acceleo [32], a reference implementation of OMG's MOFM2T standard [54] for performing model-to-text transformations. The test driver performs the transformation with the default options.

UML2OWL [25] transforms UML models to Web Ontology Language (OWL) [86] models, as described in OMG's ODM specification [55]. The model-to-model transformation is implemented based on the Atlas Transformation Language (ATL) [23]. Again, the test driver performs the transformation with the default options.

EMFCompare [22] is a standard tool for comparing and merging EMF models in the context of model versioning. The test driver computes the difference between two models provided as input and attempts to merge them into one model.

EcoreUtil [24] is a library included in EMF that provides different utility methods to modify and inspect EMF models. The test driver performs multiple of these operations on the input model, namely copy, deletion, and computation of arbitrary properties.

The selected subjects cover a wide range of different MDSE-specific tasks and applications, including model-to-model and model-to-code transformations, model validation, model comparison and merging, as well as a basic library for implementing modeling tools. In addition, our subject selection includes both generic and language-specific tools, where the latter are based on an EMF-based implementation of the UML meta-model. All of our selected subjects are still under active development, heavily used within the MDSE community, and they have a development history of more than a decade. Thus, we can assume a certain level of maturity for all of our subjects; they are unlikely to comprise shallow bugs, rendering fault finding through fuzzing into a non-trivial task.

4.1.2 Configuration of the Fuzzer Suite. Although our fuzzer suite could work with any EMF-based meta-model for fuzz testing the generic tools, in our experiments, we stick to using the UML meta-model for this purpose to enable a better comparison between the different experimental subjects. Furthermore, the UML is a well-known and comprehensive modeling language which is defined by a meta-model whose complexity is comparable to other proprietary modeling languages and tools such as Matlab/Simulink or ETAS/Ascet.

As for algorithmic parameters, we set a maximum target size of 500 for the generated models as a middle ground between models that are too simple to exercise interesting program functionality and models that are too large to be generated within a reasonable time for all approaches. However, Zest may dynamically adjust the actual target count as part of its parameter search algorithm. For the mutation-based fuzzers, we limit the size of their input queue to store at most 50 models that executed new program parts.

4.1.3 Baseline Selection. We have selected Zest [61] and the EMF Random Instantiator [5] (for brevity, from now on referred to as AtlanMod generator due to its historical origin, see Section 2.2.3) as baseline for our comparative benchmark. Each of them represents a state-of-the-art tool of the two fields of research, structure-aware fuzzing and automated model generation, which we integrate in this paper.

4.1.4 Number of Trials and Repetitions. Due to the fundamental random nature of fuzzing (most notably when performing mutations), the performance of the fuzzer can vary across multiple runs. To account for this, 30 trials are performed for each subject. Results are reported in terms of mean and standard deviation. Furthermore, statistical tests are performed in order to determine whether any observed differences are statistically significant. For randomized testing algorithms, Arcuri and Briand [3] suggest to use the Mann-Whitney U test (also known as the Wilcoxon rank-sum test).

Another important question is how long to run the fuzzer on each subject, since the performance of the fuzzer may vary over time as well. For instance, certain program regions may only be accessible if the input meets highly specific criteria. As a consequence, as long as the fuzzer is unable to generate such inputs, these regions (possibly containing many paths) cannot be explored. Commonly used timeouts in research range between a few hours [15, 61] to 1 day [9, 43, 65]. In this evaluation, a timeout of 1 hour is used, as preliminary test runs revealed that map coverage (see below) plateaued for almost all test subjects after this time.

4.1.5 Instrumentation. In order to accurately measure how effectively each approach can cover the core functionalities of the different subjects, we only include the relevant packages in the bytecode instrumentation to determine the covered branches. That is, we exclude coverage results from common external libraries such as the Eclipse Modeling Framework. Otherwise, due to the large size of the libraries compared to the benchmark subjects, the coverage results would not accurately reflect the coverage of the subjects, but rather those of the libraries.

4.1.6 Hardware. All experiments were conducted on a machine with an Intel(R) Xeon(R) E7-4880 2.5GHz CPU and 1TB of RAM running openSUSE Leap 15.

4.2 Research Questions

Using the described experimental setup, we strive to answer the following research questions to evaluate the suitability of MoFuzz to test model-driven software development tools:

- **RQ1 (Coverage):** Can MoFuzz improve the code coverage when fuzzing MDSE tools compared to the two baseline approaches? (Section 4.3.1)

- **RQ2 (Fault Finding):** Can MoFuzz improve the fault finding capabilities when fuzzing MDSE tools compared to the two baseline approaches? (Section 4.3.2)

4.3 Experimental Results

4.3.1 RQ1 Coverage Results. To assess coverage, we use the *map coverage* metric that is provided by the JQF framework [60]. Due to the enormous number of program executions during a fuzzing trial, computing the coverage for each run should be as lightweight as possible. Therefore, JQF efficiently maps each branch executed during a fuzzing run to a specific entry in a fixed-size coverage map and reports the fraction of non-zero entries as the map coverage. The map coverage can thus be interpreted as an approximate proxy for the actual branch coverage. Since the size of the map is usually much larger than the total branch count, the map coverage typically corresponds to a small fraction of the actual branch coverage. In particular, the coverage map has by default a size of $2^{16} - 1$, similar to the size of the coverage map used by AFL [1]. Full branch coverage of a program that contains 10,000 branches would thus translate to about 15% map coverage. As a result, in terms of our experiments, relative differences between the results are more important to judge about the fuzzers' performances than the absolute percentages reached by the tools.

Figure 3 plots the map coverage over time for each subject and technique, where each plot shows the average map coverage over the 30 trials at each time point. The overall results are reported in Table 1, which describes the average final coverage achieved after the 1h timeout as well as the standard deviation. The results indicate that for all benchmark subjects, except for UML2Java, one or more of our proposed techniques was able to significantly outperform the baseline approaches (highlighted cells). On the contrary, all approaches could perform equally well on UML2Java, which is a rather simple subject compared the other ones.

Overall, the AtlanMod generator performed the worst in terms of coverage. This was also expected due to the complete random generation approach. Further, while Zest is able to effectively guide the generator to incrementally increase coverage, it still falls short compared to our techniques and also produces less stable results. On the other hand, MoFuzz-emf-modelgen and MoFuzz-cgf-emfedit performed the best out of all techniques, with MoFuzz-emf-modelgen generally having the fastest and highest overall map coverage. While MoFuzz-cgf-cpeo was able to outperform AtlanMod for most subjects, the improvement was typically not enough to surpass Zest as well. One possible reason for this might be that the model generation process is slightly slower for MoFuzz-cgf-cpeo. The graph patterns comprised by most Henshin rules implementing CPEOs are considerably larger than the rule patterns used by MoFuzz-emf-modelgen, which leads to longer matching times when the rules are applied.

Based on these results we can conclude that MoFuzz can improve code coverage when fuzzing MDSE tools compared to baseline methods.

[1] AFL uses a coverage map size of 2^{16}, which has empirically been found to be a good trade-off between precision and performance [93]. In JQF, this value is adjusted to further reduce collisions.

4.3.2 RQ 2: Crashes Found. Table 2 presents the crashes that we have encountered over the course of our experimental evaluation. If one of the approaches was able to trigger the crash, we report the mean time (± standard deviation) to find the crash as well as how reliably the crash could be exposed. The reliability indicates the fraction of trials where the approach was able to identify the crash at least once.

Compared to the baseline approaches, in 11 out of 12 crashes one of our techniques always either: (1) triggered the crash as reliably as the baseline, (2) triggered the crash with a higher reliability, or (3) exposed a crash that could not even be found by any of the baseline methods. The only exception is EcoreUtil-1, which Zest can expose more reliably, but on average takes significantly longer to do so. In particular, for all crashes but EMFCompare-3, at least one of our proposed techniques was able to find the crash in a significantly faster time compared to the baseline methods.

Overall, the MoFuzz fuzzer suite found 12 crashes, while the combined baseline was only able to expose a subset of it (8 crashes), as illustrated in Figure 4. While some of the discovered crashes may have the same exception type (e.g., the ClassCastExceptions and StackOverflowErrors), each of our individual fuzzers was also able to trigger a unique crash that has not been covered by any other method (UML2Java-1, EMFCompare-2, and EMF2Graphviz-3).

Therefore, the results indicate that our proposed techniques are indeed performing better in terms of bug finding capabilities compared to the baseline methods.

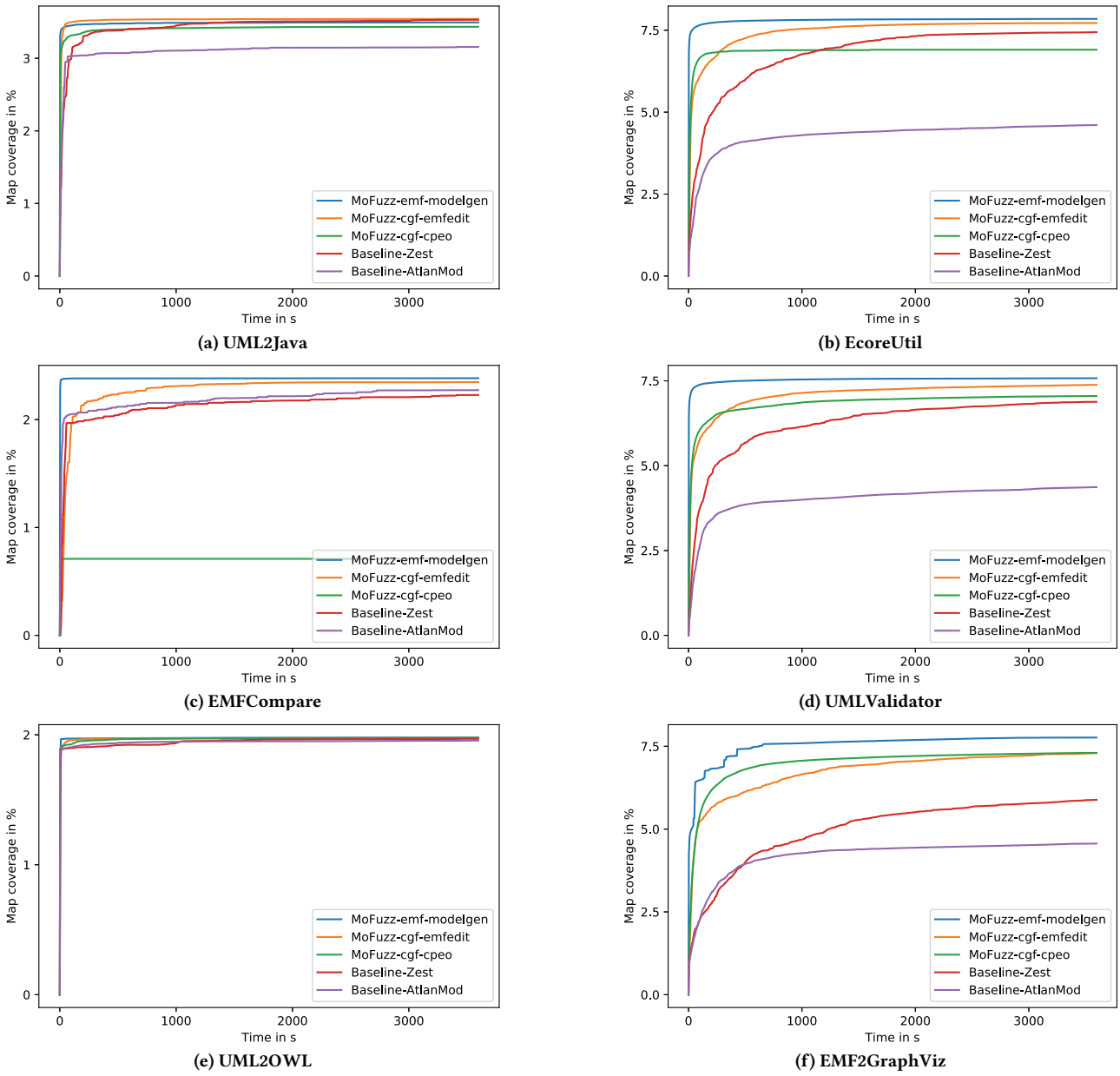
4.4 Qualitative Analysis of Selected Crashes

To identify if MoFuzz has found real bugs, we have qualitatively analyzed them and report some details below.

4.4.1 RuntimeException in UML2OWL. This crash occurs when the model contains an object of the meta-class `Slot` that does not have a value for the `DefiningFeature` reference. The model transformation attempts to access this field without validating its presence beforehand. Thus, a straightforward fix would be to validate the model with the UMLValidator before actually executing the transformation. While this error has been triggered by each approach, it is not trivial as the random AtlanMod generator was able to expose the crash with a reliability of only 60%.

4.4.2 ClassCastException in UMLValidator. This exception can be triggered by generating a model that has the following properties: (a) The model contains a `ProtocolStateMachine` object P, which itself contains a `FunctionBehavior` object F, (b) F has a cross-tree reference of type `RedefinedClassifier` to P, and (c) P has a cross-tree reference of type `OwnedBehavior` to F.

The validation of the `FunctionBehavior` instance F includes validating the `RedefinedClassifier` reference, which results in the `ClassCastException`. Triggering this crash thus requires (i) creating specific meta-class instances in a particular hierarchy and (ii) setting particular cross-tree references to the right targets. Random generation methods are likely to miss this crash as the specific meta-classes are not trivial to reach during model generation. On the other hand, the Zest-guided generator takes significantly more



(a) UML2Java (b) EcoreUtil (c) EMFCompare (d) UMLValidator (e) UML2OWL (f) EMF2GraphViz

Figure 3: Average map coverage over time (30 trials, 1h timeout). Map coverage is calculated as the number of non-zero entries in the coverage map with a fixed size of $2^{16} - 1$. A map coverage of 1% thus corresponds roughly to 655 covered branches.

Subject	Baseline approaches		MoFuzz Fuzzer Suite		
	Baseline-AtlanMod	Baseline-Zest	MoFuzz-emf-modelgen	MoFuzz-cgf-emfedit	MoFuzz-cgf-cpeo
UML2Java	3.15(±0.02)	3.53(±0.02)	3.49(±0.03)	3.54(±0.01)	3.43(±0.01)
EcoreUtil	4.61(±0.10)	7.44(±0.51)	**7.84(±0.03)**	**7.72(±0.36)**	6.90(±0.16)
EMFCompare	2.27(±0.09)	2.23(±0.15)	**2.38(±0.01)**	**2.35(±0.01)**	0.71(±0.78)
UMLValidator	4.37(±0.14)	6.89(±0.73)	**7.58(±0.09)**	**7.38(±0.29)**	7.05(±0.05)
UML2OWL	1.955(±0.007)	1.969(±0.007)	**1.979(±0.003)**	**1.974(±0.005)**	**1.977(±0.006)**
EMF2GraphViz	4.56(±0.10)	5.88(±0.82)	**7.76(±0.20)**	**7.29(±0.30)**	**7.30(±0.12)**

Table 1: Average map coverage after 1h timeout over 30 trials (± standard deviation). Highlighted values indicate significant improvements over both baseline approaches according to the Mann-Whitney U test ($\alpha = .05$).

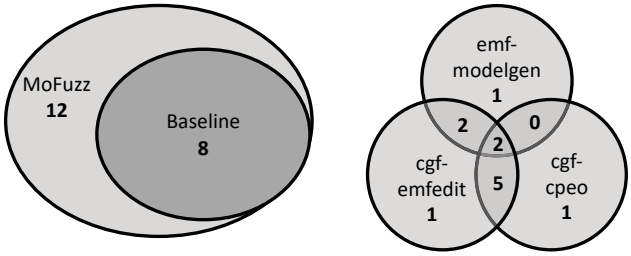


Figure 4: The number of crashes found by each approach. Left: The MoFuzz fuzzer suite triggered all crashes found by the baseline, plus additional four crashes. Right: Comparision of individual MoFuzz fuzzers, each of them exposes a unique crash.

time to reveal the crash as its mutations usually modify the model more drastically compared to our targeted mutation operators.

4.5 Threats to Validity

Internal Validity. Threats to internal validity are related to potential systematic errors during the performed experiments. To counteract these threats, we have carefully designed our experiments and followed the experimental guidelines by Klees et al. [41]. The main threat to the internal validity of the results (i.e., the degree to which possible sources of systematic errors can be ruled out) is the selection of experimental parameters. This includes parameters which have been chosen heuristically based on preliminary test runs, including the duration of each fuzzing run as well as the model generator configuration parameters. In particular, our selected target model size of 500 elements may have affected the experimental results, and much larger models might allow more complex crashes to be exposed. Further, the baseline tools have been executed using their default configurations only, though alternative configurations could have produced different results. However, a detailed parameter sensitivity analysis is out of the scope of this paper. Another common threat to internal validity for fuzzing experiments is that they might be influenced by random selections, e.g., when choosing mutation operations or graph grammar rules. While this has been addressed by running 30 runs for each subject, it cannot be guaranteed that all approaches are equally robust to the effect of such selections. The experimental data collection and analysis methods can also pose a threat to internal validity. However, the whole process has been fully automated, and all results have been calculated using well-known libraries for scientific computing, including NumPy [56] and SciPy [81].

Construct Validity. Our experiments rely on evaluation metrics that may affect the *construct validity* of the results. For the first research question, we used the map coverage provided by JQF for measuring the coverage of the MDSE tools under test. This metric can be less accurate than branch coverage due to the possibility of collisions in the coverage map. However, for all benchmark subjects, less than 10% (about 6500 branches) of the total coverage map was actually utilized, making a significant influence from collisions unlikely. Additionally, the usage of coverage metrics in general is questionable, since the goal of fuzzing tools is to find bugs. This is also why we focussed our evaluation more on the second research question that analyses the bug finding capability. However, for completeness reasons, we report also the coverage results, similar to other studies [9, 10, 43, 45, 61]. A final problem with construct validity is that the experiments just compare the performance of the tools and not their underlying concepts, which is an inherent problem of experimental evaluations. However, at least some of the tools used in our experiments are built on top of each other, and this is where we could reliably isolate the effects of the individual conceptual contributions.

External Validity. To evaluate threats to external validity we have to answer the question whether our results are generalizable to other MDSE tools. We choose EMF2GV [12], UMLValidator [21], UML2Java [53], UML2OWL [25], EMFCompare [22], and EcoreUtils [24] as subjects because of their widespread adoption in the MDSE community. The tools are long-living and mature, and they cover a variety of different applications within MDSE. Based on this diversity of these tools, we argue that the results will be transferable to tools sharing a similar level of maturity, where most of the bugs have been already discovered during their long usage. However, we cannot claim that the results are also transferable to immature prototype tools that contain plenty of shallow bugs. Here, our hypothesis is that traditional fuzzing and model generator approaches detect these shallow bugs as fast as MoFuzz.

5 CONCLUSION AND FUTURE WORK

In this paper, we presented MoFuzz, a fuzzer suite which is tailored to the problem of fuzz testing tools in model-driven software development (MDSE). The suite includes a graph grammar-based fuzzer and two variants of a coverage-guided mutation-based fuzzer. The intuition behind the former is the attempt to cover different aspects of the input domain as fast as possible, while the latter two aim to incrementally evolve input models to exercise deeper paths in the MDSE tool under test. The main variation point between the two mutation-based fuzzers is the set of change operations used for model modification, with the main motivation of exploring the trade-off between efficiency of mutations and the level of consistency which is guaranteed to be preserved by the model mutations.

Our experimental results are twofold. First, we could show that all fuzzers within the MoFuzz suite can significantly outperform existing standard fuzzers and model generators w.r.t. to covering the MDSE tool under test and their fault detection capabilities. This demonstrates the benefits of integrating these two kinds of technologies for the sake of fuzz testing MDSE tools, which was the main research question guiding our work. Second, our experimental results demonstrate that the fuzzing approaches within MoFuzz indeed come with their own strengths and weaknesses. Thus the fuzzers complement each other, forming a fuzzer suite for testing MDSE tools which can contribute to maturing of modeling tools, which is still considered as one of the biggest obstacles for adopting MDSE in practice.

Future work could focus on further enhancing the MoFuzz algorithms. For example, hybrid approaches such as Driller [75] or SymFuzz [15] effectively combine fuzzing with symbolic execution in order to find deeper bugs. Seed generation methods, such as Skyfire [82] or Orthrus [71] might be able to provide MoFuzz with

Crash ID	Exception	Approach	Average Time to Find	Reliability
UML2Java-1	StackOverflowError	MoFuzz-cgf-cpeo	**2155.82s** (±0*s*)	3%
EcoreUtil-1	StackOverflowError	MoFuzz-emf-modelgen	1968.94s (±608.51*s*)	23%
		MoFuzz-cgf-emfedit	**125.22s** (±91.51*s*)	83%
		MoFuzz-cgf-cpeo	381.13s (±0*s*)	3%
		Baseline-Zest	538.41s (±383.26*s*)	97%
EcoreUtil-2	ClassCastException	MoFuzz-cgf-emfedit	**49.95s** (±22.93*s*)	100%
		MoFuzz-cgf-cpeo	185.61s (±36.35*s*)	40%
		Baseline-Zest	181.03s (±202.65*s*)	97%
EMFCompare-1	UnsupportedOperationException	MoFuzz-emf-modelgen	**238.25s** (±23.73*s*)	100%
		MoFuzz-cgf-emfedit	**2060.57s** (±964.59*s*)	30%
EMFCompare-2	IndexOutOfBoundsException	MoFuzz-cgf-emfedit	**1533.42s** (±758.23*s*)	57%
EMFCompare-3	IllegalStateException	MoFuzz-cgf-emfedit	20.64s (±4.00*s*)	100%
		MoFuzz-cgf-cpeo	83.41s (±27.07*s*)	100%
		Baseline-Zest	6.40s (±2.73*s*)	100%
		Baseline-AtlanMod	25.18s (±7.01*s*)	100%
UMLValidator-1	StackOverflowError	MoFuzz-emf-modelgen	1337.24s (±907.88*s*)	17%
		MoFuzz-cgf-emfedit	**126.06s** (±92.31*s*)	100%
		Baseline-Zest	528.82s (±327.61*s*)	93%
UMLValidator-2	ClassCastException	MoFuzz-cgf-emfedit	**53.28s** (±26.39*s*)	100%
		MoFuzz-cgf-cpeo	824.57s (±587.26*s*)	100%
		Baseline-Zest	217.43s (±210.91*s*)	100%
UML2OWL-1	RuntimeException	MoFuzz-emf-modelgen	**240.02s** (±25.11*s*)	100%
		MoFuzz-cgf-emfedit	**169.45s** (±145.89*s*)	100%
		MoFuzz-cgf-cpeo	**109.84s** (±44.20*s*)	100%
		Baseline-Zest	1034.93s (±697.49*s*)	93%
		Baseline-AtlanMod	1218.62s (±1004.73*s*)	60%
EMF2Graphviz-1	StackOverflowError	MoFuzz-cgf-emfedit	**872.36s** (±590.21*s*)	90%
		MoFuzz-cgf-cpeo	2294.45s (±1251.17*s*)	7%
		Baseline-Zest	1767.62 (±1091.29*s*)	43%
EMF2Graphviz-2	ClassCastException	MoFuzz-cgf-emfedit	**191.68s** (±147.91*s*)	100%
		MoFuzz-cgf-cpeo	1193.68s (±726.91*s*)	73%
		Baseline-Zest	1076.54s (±938.23*s*)	87%
EMF2Graphviz-3	RuntimeException	MoFuzz-emf-modelgen	**544.19s** (±966.30*s*)	37%

Table 2: Average time to find each crash discovered in the experiments (calculated over the crash-triggering runs). Highlighted values represent significant improvements over the baseline according to the Mann-Whitney U test ($\alpha = .05$). The reliability is the percentage of runs (out of 30 trials) where the crash was triggered, rounded to the closest integer.

a high quality seed corpus. A more straightforward alternative to these approaches would be to further tweak our instance generation algorithms to focus on the generation of unexpected and slightly malformed inputs, e.g., by deliberately violating multiplicity constraints, ignoring required references, setting invalid attribute values, etc. For example, the model generation and mutation processes of MoFuzz could be guided by attaching probabilities to the model transformation rules as applied in the domain of grammar-based fuzzing [20, 73]. We would also like to study potential benefits of integrating our generator- and mutation-based fuzzers more tightly. On the one hand, since the EMF Model Generator [50] which is used as a basis of our generator-based fuzzer can work with seed input models, it could be integrated with the feedback loop of the JQF framework to aim at a more coverage-guided behavior. On the other hand, it could be used to generate seed inputs for the two mutation-based fuzzers, which currently work as a two-phase process creating their own seed inputs used for mutation. Moreover, we intend to explore the fuzzing characteristics of the other generation strategies (with different configurations) provided by the EMF Model Generator for generating diverse and huge models. Finally, we intend to broaden our experiments to include further MDSE tools working with different kinds of models in order to strengthen the external validity of our experimental results.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (DFG), projects "Meta-Modeling and Graph Grammars: Generating Development Environments for Modeling Languages" (grant no. HA 2936/4-2 and TA 294/13-2) and "ProCI: Process Conformance under Incomplete Information" (grant no. GR 3634/7-1).

REFERENCES

[1] Vlad Acretoaie and Harald Störrle. 2014. Hypersonic: Model analysis and checking in the cloud. In *2nd Workshop on Scalability in Model Driven Engineering*.

[2] Dave Aitel. 2002. The advantages of block-based protocol analysis for security testing. *Immunity Inc., February* (2002).

[3] Andrea Arcuri and Lionel C. Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 1–10. https://doi.org/10.1145/1985793.1985795

[4] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. 2010. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Proc. MODELS*. Springer, 121–135.

[5] AtlanMod. 2015. EMF Random Instantiator. https://github.com/atlanmod/mondo-atlzoo-benchmark/tree/master/fr.inria.atlanmod.instantiator/. Accessed: August 26, 2020.

[6] Amine Benelallam, Massimo Tisi, István Ráth, Benedeck Izso, and Dimitrios S Kolovos. 2014. Towards an open set of real-world benchmarks for model queries and transformations. In *BigMDE@ STAF*.

[7] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. 2012. Formal Foundation of Consistent EMF Model Transformations by Algebraic Graph Transformation. *SoSyM* 11, 2 (2012), 227–250.

[8] Robert Bill, Martin Fleck, Javier Troya, Tanja Mayerhofer, and Manuel Wimmer. 2017. A local and global tour on MOMoT. *SoSyM* (2017), 1–30.

[9] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[11] Alexander Boll and Timo Kehrer. 2020. On the Replicability of Experimental Tool Evaluations in Model-based Development. In *Intl. Conference on Systems Modelling and Management*. to appear.

[12] Jean-François Brazeau. 2011. EMF To Graphviz (emf2gv). http://emftools.tuxfamily.org/wiki/doku.php?id=emf2gv:start. Accessed: August 26, 2020.

[13] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. 2006. Metamodel-based test generation for model transformations: an algorithm and a tool. In *Symp. on Software Reliability Engineering*. 85–94.

[14] Caca Labs. 2015. zzuf - multiple purpose fuzzer. http://caca.zoy.org/wiki/zzuf. Accessed: August 26, 2020.

[15] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 725–741. https://doi.org/10.1109/SP.2015.50

[16] Shafiul Azam Chowdhury, Taylor T Johnson, and Christoph Csallner. 2016. CyFuzz: A differential testing framework for cyber-physical systems development environments. In *International Workshop on Design, Modeling, and Evaluation of Cyber Physical Systems*. Springer, 46–60.

[17] Shafiul Azam Chowdhury, Soumik Mohian, Sidharth Mehra, Siddhant Gawsane, Taylor T Johnson, and Christoph Csallner. 2018. Automatically finding bugs in a commercial cyber-physical system development tool chain with SLforge. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*. 981–992.

[18] Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T Johnson, and Christoph Csallner. 2020. SLEMI: Equivalence Modulo Input (EMI) Based Mutation of CPS Models for Finding Compiler Bugs in Simulink. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020)*.

[19] Gwendal Daniel, Gerson Sunyé, Amine Benelallam, and Massimo Tisi. 2014. Improving memory efficiency for processing large-scale models. In *BigMDE*.

[20] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. 2020. Evolutionary Grammar-Based Fuzzing. In *Proceedings of the 12th Symposium on Search-Based Software Engineering (SSBSE 2020)*.

[21] Eclipse Foundation. 2018. UML2. https://wiki.eclipse.org/MDT/UML2. Accessed: August 26, 2020.

[22] Eclipse Foundation. 2019. EMFCompare. https://www.eclipse.org/emf/compare/. Accessed: August 26, 2020.

[23] Eclipse Foundation. 2020. Atlas Transformation Language (ATL). https://www.eclipse.org/atl/. Accessed: August 26, 2020.

[24] Eclipse Foundation. 2020. Eclipse Modeling Framework. https://www.eclipse.org/modeling/emf/. Accessed: August 26, 2020.

[25] Eclipse Foundation. 2020. UML to OWL Transformation. https://www.eclipse.org/atl/atlTransformations/#UML2OWL. Accessed: August 26, 2020.

[26] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. 2006. *Fundamentals of Algebraic Graph Transformation*. Springer.

[27] Karsten Ehrig, Jochen Malte Küster, and Gabriele Taentzer. 2009. Generating instance models from meta models. *SoSyM* 8, 4 (2009), 479–500.

[28] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. 2001. Graphviz - Open Source Graph Drawing Tools. In *International Symposium on Graph Drawing*.

[29] Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. 2016. Security Testing: A Survey. Vol. 101. 1–51. https://doi.org/10.1016/bs.adcom.2015.11.003

[30] Adel Ferdjoukh, Anne-Elisabeth Baert, Annie Chateau, Rémi Coletta, and Clémentine Nebut. 2013. A CSP approach for metamodel instantiation. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. IEEE, 1044–1051.

[31] Franck Fleurey, Jim Steel, and Benoit Baudry. 2004. Validation in model-driven engineering: testing model transformations. In *Proc. Intl. Workshop on Model, Design and Validation*. IEEE, 29–40.

[32] Eclipse Foundation. 2019. Acceleo. https://www.eclipse.org/acceleo/. Accessed: August 26, 2020.

[33] Antonio Garmendia, Antonio Jiménez-Pastor, and Juan de Lara. 2015. Scalable model exploration through abstraction and fragmentation strategies. In *CEUR Workshop Proceedings*. CEUR-WS.

[34] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 206–215. https://doi.org/10.1145/1375581.1375607

[35] Carlos A González, Fabian Büttner, Robert Clarisó, and Jordi Cabot. 2012. EMFtoCSP: A tool for the lightweight verification of EMF models. In *2012 First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*. IEEE, 44–50.

[36] Google. 2020. FuzzBench: Fuzzer Benchmarking As a Service. https://github.com/google/fuzzbench. Accessed: August 26, 2020.

[37] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. USENIX, Bellevue, WA, 445–458.

[38] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (2002), 256–290.

[39] Timo Kehrer, Michaela Rindt, Pit Pietsch, and Udo Kelter. 2013. Generating Edit Operations for Profiled UML Models. In *Proceedings of the Intl. Workshop on Models and Evolution (CEUR Workshop Proceedings)*, Vol. 1090. 30–39.

[40] Timo Kehrer, Gabriele Taentzer, Michaela Rindt, and Udo Kelter. 2016. Automatically Deriving the Specification of Model Editing Operations from Meta-Models. In *Proc. ICMT*. 173–188.

[41] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[42] Maher Lamari. 2007. Towards an automated test generation for the verification of model transformations. In *Proc. ACM Symposium on Applied Computing*. ACM, 998–1005.

[43] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 475–485. https://doi.org/10.1145/3238147.3238176

[44] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (2018), 6. https://doi.org/10.1186/s42400-018-0002-y

[45] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 627–637. https://doi.org/10.1145/3106237.3106295

[46] Matthew J McGill, RE Kurt Stirewalt, and Laura K Dillon. 2009. Automated test input generation for software that consumes ORM models. In *OTM Confederated Intl. Conferences*. Springer, 704–713.

[47] Jacqueline McQuillan and James Power. 2008. A metamodel for the measurement of object-oriented systems: An analysis using Alloy. In *Intl. Conf. on Software Testing, Verification, and Validation*. IEEE, 288–297.

[48] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279

[49] Alix Mougenot, Alexis Darrasse, Xavier Blanc, and Michèle Soria. 2009. Uniform random generation of huge metamodel instances. In *European Conf. on Model Driven Architecture-Foundations and Applications*. Springer, 130–145.

[50] Nebras Nassar, Jens Kosiol, Timo Kehrer, and Gabriele Taentzer. 2020. Generating Large EMF Models Efficiently - A Rule-Based, Configurable Approach. In *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science)*, Heike Wehrheim and Jordi Cabot (Eds.), Vol. 12076.

Springer, 224–244. https://doi.org/10.1007/978-3-030-45234-6_11
[51] Nebras Nassar, Jens Kosiol, and Hendrik Radke. 2017. Rule-based Repair of EMF Models: Formalization and Correctness Proof. In *Electronic Pre-Proc. Intl. Workshop on Graph Computation Models*.
[52] Nebras Nassar, Hendrik Radke, and Thorsten Arendt. 2017. Rule-Based Repair of EMF Models: An Automated Interactive Approach. In *Proc. ICMT*. 171–181.
[53] Obeo. 2020. UML to Java Generator. https://marketplace.eclipse.org/content/uml-java-generator. Accessed: August 26, 2020.
[54] Object Management Group (OMG). 2008. *MOF Model to Text Transformation Language, v1.0*. Standard. https://www.omg.org/spec/MOFM2T/1.0/PDF
[55] Object Management Group (OMG). 2014. *Ontology Definition Metamodel*. Standard. https://www.omg.org/spec/ODM/1.1/PDF
[56] Travis E. Oliphant. 2006. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA.
[57] OMG 2014. *Object Constraint Language*. OMG. http://www.omg.org/spec/OCL/
[58] OMG 2015. *XML Metadata Interchange*. OMG. https://www.omg.org/spec/XMI/
[59] OMG 2016. *OMG Meta Object Facility (MOF). Version 2.5.1*. OMG. http://www.omg.org/spec/MOF/
[60] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 398–401.
[61] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576
[62] Peach Tech. 2020. Peach Fuzzer Platform. https://www.peach.tech/products/peach-fuzzer/peach-platform/. Accessed: August 26, 2020.
[63] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru R. Căciulescu, and Abhik Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2019), 1–17.
[64] Pit Pietsch, Hamed Shariat Yazdi, and Udo Kelter. 2011. Generating realistic test models for model processing tools. In *Proc. ASE*. IEEE CS, 620–623.
[65] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/
[66] Michaela Rindt, Timo Kehrer, and Udo Kelter. 2014. Automatic Generation of Consistency-Preserving Edit Operations for MDE Tools. In *Proceedings of the Demonstrations Track of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems*.
[67] Jesse Ruderman. 2007. Introducing jsfunfuzz. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/. Accessed: August 26, 2020.
[68] Oszkár Semeráth, Aren A. Babikian, Sebastian Pilarski, and Dániel Varró. 2019. VIATRA solver: a framework for the automated generation of consistent domain-specific models. In *Proc. ICSE*. IEEE/ACM, 43–46.
[69] Oszkár Semeráth, András Szabolcs Nagy, and Dániel Varró. 2018. A Graph Solver for the Automated Generation of Consistent Domain-specific Models. In *Proc. ICSE*. ACM, 969–980.
[70] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. 2009. Automatic model generation strategies for model transformation testing. In *Proc. ICMT*. 148–164.
[71] Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. 2017. Static Program Analysis as a Fuzzing Aid. In *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings (Lecture Notes in Computer Science)*, Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis (Eds.), Vol. 10453. Springer, 26–47. https://doi.org/10.1007/978-3-319-66332-6_2
[72] Sohil Lal Shrestha, Shafiul Azam Chowdhury, and Christoph Csallner. 2020. DeepFuzzSL: Generating Simulink Models with Deep Learning to Find Bugs in the Simulink Toolchain. In *Workshop on Testing for Deep Learning and Deep Learning for Testing (DeepTest 2020)*.
[73] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2020. Inputs from Hell: Learning Input Distributions for Grammar-Based Test Generation. *IEEE Transactions on Software Engineering* (2020).
[74] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework* (2 ed.). Addison Wesley, Upper Saddle River, NJ.
[75] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf
[76] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. 2017. Henshin: A usability-focused framework for emf model transformation development. In *International Conference on Graph Transformation*. Springer, 196–208.
[77] Daniel Strüber, Timo Kehrer, Thorsten Arendt, Christopher Pietsch, and Dennis Reuling. 2016. Scalability of Model Transformations: Position Paper and Benchmark Set. In *BigMDE@ STAF*. 21–30.
[78] Andreas Svendsen, Øystein Haugen, and Birger Møller-Pedersen. 2011. Synthesizing software models: generating train station models automatically. In *Intl. SDL Forum*. Springer, 38–53.
[79] Gabriele Taentzer. 2012. Instance Generation from Type Graphs with Arbitrary Multiplicities. *ECEASST* 47 (2012).
[80] Spandan Veggalam, Sanjay Rawat, István Haller, and Herbert Bos. 2016. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I (Lecture Notes in Computer Science)*, Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows (Eds.), Vol. 9878. Springer, 581–601. https://doi.org/10.1007/978-3-319-45744-4_29
[81] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* (2020).
[82] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 579–594. https://doi.org/10.1109/SP.2017.23
[83] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 724–735. https://doi.org/10.1109/ICSE.2019.00081
[84] Junhua Wang, Soon-Kyeong Kim, and David Carrington. 2008. Automatic generation of test models for model transformations. In *Australian Conf. on Software Engineering*. IEEE, 432–440.
[85] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. 2013. Industrial adoption of model-driven engineering: Are the tools really the problem?. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 1–17.
[86] World Wide Web Consortium (W3C). 2004. *OWL Web Ontology Language Reference*. Specification. https://www.w3.org/TR/owl-ref/
[87] Hao Wu. 2016. An SMT-based approach for generating coverage oriented metamodel instances. *International Journal of Information System Modeling and Design (IJISMD)* 7, 3 (2016), 23–50.
[88] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532
[89] Hamed Shariat Yazdi, Lefteris Angelis, Timo Kehrer, and Udo Kelter. 2016. A framework for capturing, statistically modeling and analyzing the evolution of software models. *Journal of Systems and Software* 118 (2016), 176–207.
[90] Hamed Shariat Yazdi, Mahnaz Mirbolouki, Pit Pietsch, Timo Kehrer, and Udo Kelter. 2014. Analysis and Prediction of Design Model Evolution Using Time Series. In *Advanced Information Systems Engineering Workshops - CAiSE 2014 International Workshops (Lecture Notes in Business Information Processing)*, Vol. 178. Springer, 1–15.
[91] Hamed Shariat Yazdi, Pit Pietsch, Timo Kehrer, and Udo Kelter. 2013. Statistical analysis of changes for synthesizing realistic test models. *Software Engineering 2013* (2013).
[92] Hamed Shariat Yazdi, Pit Pietsch, Timo Kehrer, and Udo Kelter. 2015. Synthesizing realistic test models. *Computer Science-Research and Development* 30, 3-4 (2015), 231–253.
[93] Michal Zalewski. 2017. American Fuzzy Lop (AFL) - Technical Whitepaper. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: August 26, 2020.
[94] Michal Zalewski. 2020. American Fuzzy Lop (AFL) - a security-oriented fuzzer. http://lcamtuf.coredump.cx/afl/. Accessed: August 26, 2020.
[95] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The Fuzzing Book. In *The Fuzzing Book*. Saarland University. https://www.fuzzingbook.org/
[96] Steffen Zschaler and Lawrence Mandow. 2016. Towards model-based optimisation: Using domain knowledge explicitly. In *Software Technologies: Applications and Foundations*. Springer, 317–329.