Model-Driven Fault Injection in Java Source Code

Elder Rodrigues Jr.
Universidade Estadual de Campinas
Campinas, SP, Brazil
elder.junior@students.ic.unicamp.br

Leonardo Montecchi
Universidade Estadual de Campinas
Campinas, SP, Brazil
leonardo@ic.unicamp.br

Andrea Ceccarelli
Università degli Studi di Firenze
Firenze, Italy
andrea.ceccarelli@unifi.it

***Abstract*—The injection of software faults in source code requires accurate knowledge of the programming language, both to craft faults and to identify injection locations. As such, fault injection and code mutation tools are typically tailored for a specific language and have limited extensibility. In this paper we present a model-driven approach to craft and inject software faults in source code. While its concrete application is presented for Java, the workflow we propose does not depend on a specific programming language. Following Model-Driven Engineering principles, the faults and the criteria to select injection locations are described using structured, machine-readable specifications based on a domain-specific language. Then, automated transformations craft artifacts based on OCL and Java, which represent the faults to be injected and are able to select the candidate injection locations. Finally, artifacts are executed against the target source code, performing the injection in the desired locations. We devise a supporting tool and exercise the approach injecting 13 different kinds of software faults in the Java source code of six different projects.**

***Index Terms*—Software faults, fault libraries, metamodel, OCL, code patterns, Java.**

I. Introduction

It is several years that software fault injection, i.e., the deliberate injection of software faults [31], is a well-established approach to evaluate the dependability of computer systems [22]. Its growth in relevance has been also pushed by the continuous increase of software size and complexity, which makes software faults and software-related accidents more likely to occur. It is in fact acknowledged that software faults are a relevant threat to the dependability of computer-based systems and a major cause of economical losses, with an estimation of 3.6 billion people impacted by software faults and USD 1.7 trillion revenue lost in 2017 [43]. It is then clear that software fault injection, as a mean for fault removal and fault forecasting [4], is a necessary technique.

Several fault injection approaches and tools have been proposed. They differ on multiple aspects, ranging from the scope of fault injection, the target system, the injection approach, or the specification of faults themselves. Amongst the different classes of fault injection approaches, in this paper we focus on the *injection of code changes*, which was developed to closely emulate software faults by introducing wrong code that mimics the most typical programming bugs [31].

This work has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement No 823788. This work has received funding from the São Paulo Research Foundation (FAPESP) with grants #2018/11129-8 and #2019/06799-7.

When injecting code changes, the artificial faults introduced in the software should be realistic, i.e., similar to the real software bugs, and the criteria for the injection should depend on the target system and on the purpose of the testing campaign [25]. Therefore, several authors have analyzed which are the most common faults introduced by developers, and the support of automated tools that inject realistic faults has improved in recent years. While it would be useful to reuse existing tools to create custom faults, or to alter some of the implemented ones, this can be a hard task in practice [3]. In fact, to create new injectors for code-change faults, a strong knowledge of the language and of its low-level artifacts is needed (e.g., the abstract syntax tree or the compiler).

In this paper we present and exercise a model-driven approach to craft and inject software faults in source code. Following Model-Driven Engineering (MDE, [39]) principles, the faults and the criteria to select injection points are described using structured, machine-readable, specifications based on a domain-specific language (DSL, [44]). The advantage of this approach is that the description of the fault is expressed in a high-level language, tailored do the domain of fault injection, and it does not require knowledge of the target application, its programming language, or low-level artifacts. Once transformations have been defined, any fault that can be expressed with the DSL can be automatically injected, without the need to write additional code. Also, faults are described in an abstract way, and thus they can potentially be injected in code written in any programming language, by defining appropriate model transformations.

More specifically, we provide the following contributions: i) we devise an MDE-based methodology for describing and managing faults to be injected as structured specifications; ii) we define a domain-specific language to realize such methodology; iii) we implement transformations and build a tool in order to apply the whole methodology for software programs written in Java; and iv) as case study, we use our DSL to specify a fault library from the literature, and we apply our methodology on six different Java projects, in which we inject approximately 50,000 faults in the corresponding injection points. All the source code, models, and transformations are publicly available at [37].

The rest of the paper is organized as follows. First, Section II presents background notions, and Section III describes the related work. Then, Section IV illustrates our methodology, while Section V details our domain-specific language and

transformations. Section VI details the proposed workflow through an exemplary execution, while Section VII presents the case study. Finally, Section VIII discusses the results, limitations, and possible extensions of this work. Section IX concludes the paper.

II. Background

A. Model Driven Engineering

Model-Driven Engineering (MDE, [39]) refers to the systematic use of models as primary artifacts throughout the engineering lifecycle. The idea is that the information should be described and managed at the most abstract level as possible, and concrete artifacts should be generated from this primary information. MDE techniques combine: i) Domain Specific Languages (DSLs, [44]), which formalize the information relevant for a certain domain; and ii) model-transformations and code generators [15], which analyze models and synthesize different kinds of artifacts, such as source code, simulators, or documentation.

One of the foundational concepts of MDE is metamodeling. A *metamodel* [13] formally defines what are the constructs that can appear in a certain class of models and their relations, that is, the abstract syntax of a language. A model is said to *conform to* a certain metamodel if it respects its abstract syntax. A model transformation receives as input a model m_a that conforms to a metamodel A, and produces as output a model m_b that conforms to a metamodel B.

The ability to automatically transform models and synthesize various artifacts helps to ensure the consistency between system requirements, specification, implementation, and evaluation models. Furthermore, MDE reduces human mistakes, by the application of state-of-the-art development practices, embedded in automated transformations.

MDE techniques are widely used in the industry practices and bring several benefits [23]. According to [45], the benefit is greater when those techniques are applied to specific parts or aspects of the system. In fact, by targeting a specific problem, (meta-)models and transformations are simpler and closer to concepts of the involved domain.

B. Fault Injection

The ultimate goal of fault injection is to test how tolerant a system or component is in the presence of faults. It can be used to quantify the ability of a system to detect faults, to recover from an unexpected failure, etc. [22]. Fault injection can refer either to the injection of faults into hardware or into software components; in this work we exploit *software fault injection* to inject code changes in software source code.

In order to execute fault injection tests, a fault injection environment is typically required, which mainly consists of the following entities [22]:

Load Generator. Generates the sequence of inputs to be applied to the target system, based on the description of the intended workload provided by the *Workload Library*.

Injector. Injects faults into the target system. The faults to be injected are provided by a *Faultload Library*.

TABLE I
Faults Model [17] to be implemented in our approach.

Fault ID	*Fault Name*
MFC	Missing Function Call
MIA	Missing If Construct Around Statements
MIEB	Missing If construct plus statements plus Else Before statement
MIFS	Missing IF construct plus Statements
MLAC	Missing AND expression in expression used as branch condition
MLOC	Missing OR expression in expression used as branch condition
MLPA	Missing Small and Localized part of the Algorithm
MVAE	Missing Variable Assignment using an expression
MVAV	Missing Variable Assignment using a value
MVIV	Missing Variable Initialization using a value
WAEP	Wrong Arithmetic expression in parameter of a function call
WPFV	Wrong Variable used in parameter of a function call
WVAV	Wrong Value Assigned to variable

Monitor. Observes the target system with the injected faults and collects the generated data.

Controller. Coordinates the other entities.

Fault injection is effective if the injected faults are realistic, because the goal is to estimate how tolerant a system is in the presence of real faults. In the literature, several authors have discussed what are the most common faults introduced by developers, and the support of automated tools that inject faults similar to real ones flourished in the last decades [16, 20, 24, 25, 29, 30, 38]. In this paper we have selected 13 faults that we will use in the experimental evaluation of our approach. Such faults are listed in Table I; they are the same originally selected by Durães *et al.* [17], based on their empirical study on the representativeness of different faults. More recent studies confirm that this fault model can still be deemed up-to-date and credible [31], and that it is substantially a subset of fault models for Java [8, 38].

C. How we will exploit background knowledge

In this work, we are proposing a MDE approach for fault injection, where faults can be specified through a DSL, that is, a language tailored to the domain of fault injection. A collection of specifications constitutes the *Faultload Library*. Once the specification is done, a model-transformation automatically generates an implementation of the needed *Injector(s)*. Given any software in the target programming language, the injector is able to i) find the possible injection points and then ii) concretely inject the faults in the identified injection points.

III. Related Work

The main contribution of this paper is to provide a framework for describing fault patterns at abstract level, from which injectors can be automatically derived, and to demonstrate its application to Java software. To the best of our knowledge there are few approaches that propose a similar contribution.

Numerous tools to inject faults in Java applications exist in the state of the art (e.g., see [28]), and the authors usually explain how to extend their tools in order to implement injectors for additional faults. However, often the knowledge of low level artifacts is still necessary. Our proposal here is to

define a language where it is possible to define fault patterns at a high level of abstraction, and then automatically derive (i.e., generate) the injector code. We generate fault-specific code that finds the candidate injection points and then injects the fault into the source code.

Few works adopt a high-level approach. Hoarau *et al.* [21] investigated the possibility of injecting software faults in Java applications using a high-level language called FAIL. However, their strategy differs from our proposal, since FAIL is used to describe fault scenarios at runtime (e.g., "do not execute the last loop of while/for statements"). Conversely, our language specifies injection points as code patterns, coupled with a sequence of actions (e.g., "find a variable declaration and delete its initialization") to directly alter the source code. Olah *et al.* [35] presented a model-based approach to specify fault injection experiments and run-time monitoring. Although their approach is based on MDE techniques, their work relies on a third-party tool to provide the functionality to find injection points and to perform the injection. Furthermore, their approach injects faults in the Java bytecode, while we operate on the source code.

Some studies use high-level languages to inject faults in non-Java systems. The work by Navat *et al.* [32] aims to automate fault injection in cyber-physical systems by integrating the injection process within an MDE design flow. However, the paper does not deal with a methodology for the characterization and description of software faults, which is instead one of the major contributions of our work. The recent work by Cotroneo *et al.* [14] adopts a similar approach to the one proposed in this paper, but it targets Python code. There are also some differences in the realization. First, our language is more expressive in the definition of injection points; for example we can define patterns across different classes or based on object-oriented concepts like inheritance and visibility modifiers. Conversely, their language appears to be more expressive concerning the actions to be executed. Also, our approach works with the abstract syntax (metamodel) and it is in principle easier to be adapted to other languages.

Several works on mutation testing and on the generation of mutants relied on the support of MDE, with the most relevant to our paper being [7, 18, 19, 40]. Despite some similarities especially in the usage of MDE principles, these works have a different purpose, and none of them target the definition of an application-independent approach to specify and inject software faults in source code as we are proposing. In fact, Wodel [18] is a DSL to generate mutants of models conforming to arbitrary metamodels; Bartel *et al.* [7] generate a fault model for the adaptation logic of adaptive systems; Simão et al. [40] exploit transformational and logical programming paradigms rather than MDE principles; and Gomez *et al.* [19] introduce a tool to generate mutation testing tool for domain-specific languages.

In the field of robustness testing, Moraes *et al.* [27] presented an extension of the UML Testing Profile [6] that includes the main concepts of robustness testing, for the purpose of improving documentation of experiments, while our work permits to craft and actually inject software bugs.

Also, a relevant body of research in dependable cyber-physical systems exploits MDE techniques and UML profiles. However, the purpose is typically to model failures and understand their propagation into the system by some kind of simulation [9, 26]. Finally, other works concentrate on model-driven fault injection in embedded systems, for example [5, 10, 12]. These works are specifically tailored to embedded systems, thus assuming domain specificities like the availability of specific input models or architectural paradigms. Our paper targets generic Java software and thus requires a more general approach.

IV. Methodology

In this section we describe the general workflow we envision for a model-driven injection of software faults, and the concrete technical workflow we adopted to realize it.

A. General Workflow

The workflow we propose for fault injection supported by MDE techniques is centered around a DSL that is used to provide specifications of the faults to be injected as structured, machine-readable, models. This follows the MDE principle that *everything is a model* [13] and that therefore everything should be treated like one.

In MDE, high-level specifications are automatically converted into concrete artifacts (e.g., source code, test cases) by model transformation and code generation algorithms. Typically, both the model and the transformations are domain-specific, that is, they embed the knowledge of a certain domain, thus allowing a greater degree of abstraction.

In our case, the low-level artifacts are the *injectors*, which are generated from fault specifications. Given a certain class of systems (e.g., those written in Java) the generation of injectors can be automated and embedded in model transformation algorithms. Changing the class of target systems, for example switching from Java to C++, requires an adaptation of the transformation algorithms. However, the specification of the faults, which are the input to the transformations, can remain unchanged. The best practices for injection in C++ code could again be embedded in the transformations.

The main benefit of this approach is that once the DSL and the transformations are defined, the injector for all the possible specifications supported by the language is automatically derived. This contrasts with common practice in the literature, in which injectors for a limited selected set of selected faults are programmed in an *ad hoc* fashion.

The workflow we propose consists of the following three macro-steps:

Specification. The first step consists in writing a fault injection specification i.e., to create structured models using the DSL (described in Section V later). In general, two pieces of information are required in a fault specification: i) identification of the injection point where the fault can be injected; and ii) the action to be performed to inject the fault e.g., removal or modification of a code element.

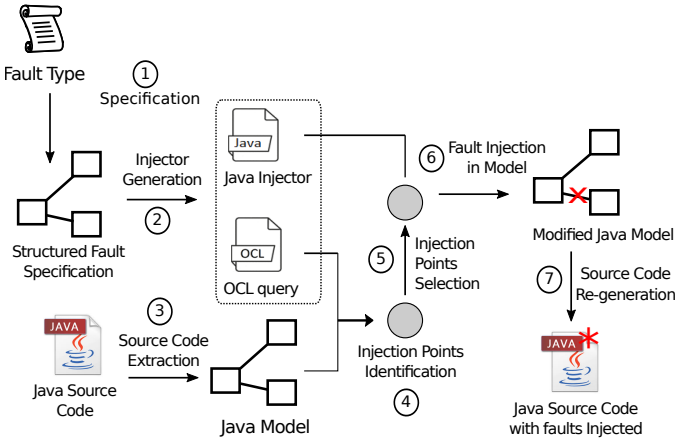

Fig. 1. Our concrete workflow for model-driven injection of software faults.

Generation. In the second step a set of model transformations processes the fault specification (in the form of a structured model), and derives the artifacts that implement the injector. That is, the second step derives a program that is able to inject the fault specified using the DSL.

Injection. Finally, the derived artifacts that constitute the injector are executed and the specified fault is injected in the target software project.

This workflow is general and it can be realized for different classes of systems. In the following we describe how we have realized it for injecting faults in Java source code.

B. Technical Workflow

To better understand how we concretely implemented the proposed workflow, we provide here more details about the artifacts generated in each step, and how the faults are actually injected into the source code. When possible, we reused existing MDE tools and frameworks, in an effort to focus on aspects purely related to fault injection.

The detailed workflow is presented in Figure 1 and is discussed in the following.

1) Fault Specification: This step corresponds to the *Specification* step of the general workflow, and it is where faults to be injected are specified.

We base our DSL on the recent work in [36], which defined the CCSL (Coding Conventions Specification Language). The original purpose of CCSL is to specify rules of coding conventions (e.g., the SEI CERT Coding Conventions [41]), and then generate checkers that are able to find violations of such rules in the source code. Using CCSL it is possible to define forbidden coding patterns by describing the involved code elements and their relations. In particular, CCSL has been applied for specifying coding rules for Java in [36].

Two pieces of information are needed to specify the injection of a fault: i) the injection points where the fault can be injected; and ii) which actions are necessary to inject the fault. For the first part we can reuse CCSL: instead of specifying patterns of violations in the source code, we use it to specify patterns of candidate injection points. Of course, additional information that is specific of the fault injection domain needs to be added to the language. How we reuse and extend CCSL is described in Section V.

2) Injector Generation: In this step the injector is automatically generated from the fault specification. The injector also adopts a model-driven approach, that is, the manipulations are made on a structured representation of the code and not on the code itself. In this work we use the Java metamodel provided by MoDisco [11], which provides a one-to-one representation of source code elements. Metamodels that can be used with other languages are available e.g., the Knowledge Discovery Metamodel (KDM [34]).

Two artifacts are generated, corresponding to the two pieces of information in the specification: an *OCL query* (`fault.ocl`), and a *Java class* (`faultAction.java`). The model-to-text generation of these two artifacts is implemented with the Acceleo framework [1].

The OCL (Object Constraint Language) [33] is a declarative language used to specify constraints in models and metamodels. Besides this original purpose, it can be used as a query language for models. The generated OCL query is used to retrieve the model elements that are possible injection points for the fault. Once these elements have been identified, the generated Java class is used to concretely apply the specified actions e.g., delete or replace an element.

3) Source Code Extraction: In order to execute the OCL query generated in the previous step, the structured model of the target source code must be available. For this we rely on the MoDisco tool [11], whose purpose is exactly to extract a detailed structured model from a Java project. It should be noted that MoDisco gives the possibility to extend its extractor to other programming languages.

4) Injection Points Identification: Once the structured model of the target project is available, the generated OCL query (`fault.ocl`) is executed on it. The set of the possible injection points is the output of this step.

5) Injection Points Selection: Once the list of possible injection points have been retrieved, one or more algorithms (strategies) are executed to select which injection points should be used by the injector. The selection strategies are not generated automatically, and they need to be previously defined.

6) Fault Injection in Model: Once the injection points have been selected, the generated Java injector is executed in order to modify the code elements at the selected injection points, and thus inject the specified fault. After this step, the extracted model of the target project is now representing the source code modified with the faults specified in Step 1.

7) Source Code Regeneration: The final step takes as input the structured model representing the modified project, and it regenerates the whole source code with the injected faults. To perform this step we use again the MoDisco tool [11].

V. Injection Specification Metamodel

As it should be clear from the previous section, the main contributions of this paper are i) the adaptation and extension

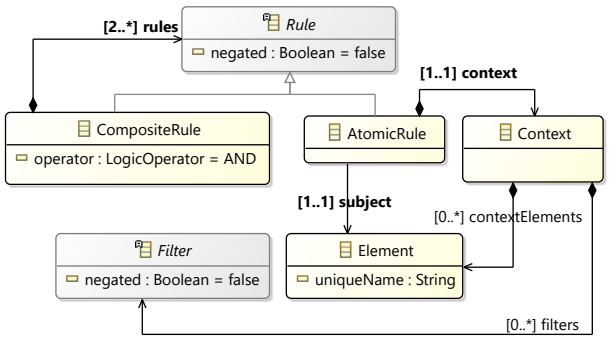

Fig. 2. CCSL core elements.

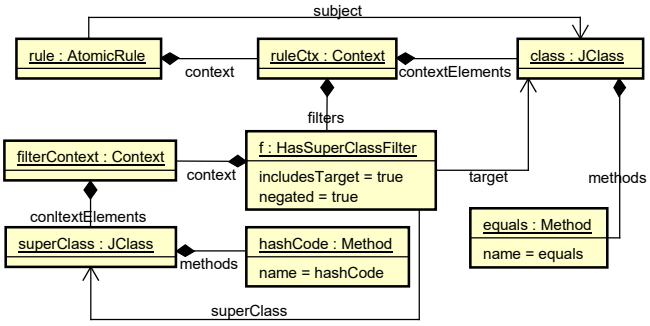

Fig. 3. MET09-J CCSL specification.

of CCSL for specifying faults to be injected, and ii) the transformations that generate injectors from such specifications. In this section we briefly recall about CCSL, we explain the extensions that we introduced, and we discuss how it can be used for the specification of faults.

A. Coding Conventions Specification Language

The CCSL has been defined using the Eclipse Modeling Framework (EMF) [42], a well-known meta-modeling framework that provides an unifying runtime layer for different MDE tools withing Eclipse.

The main element of the CCSL metamodel is the *Rule* metaclass, which represents a pattern that identifies a coding rule. A rule can be either atomic or composite, as illustrated in Figure 2. An *AtomicRule* consists of a context and a subject. The *Context* describes the pattern to be searched in the source code e.g., a class with name "Foo" that contains at least one method called as "qux". The context of a rule must contain at least one *Element* and it may contain a certain number of *Filter* instances.

The *Element* metaclass is the top of the hierarchy of classes that represent elements of the source code e.g., classes, interfaces, methods, invocations, assignments, etc. The metamodel of CCSL contains different subclasses of *Element*, which are not reproduced here for the sake of brevity. The full CCSL metamodel is available on GitHub [37]. The elements that constitute the context (and their relations) form the base for the pattern to be found in the code.

Filters are used to retain only elements of the context that fulfill specific conditions e.g., to select only classes whose name is matched by a regular expression. *Filter* is an abstract metaclass, and it is extended by several concrete filters. A filter can be *negated*, which means that only elements *not* fulfilling the filter are selected. The full CCSL metamodel contains different kind of filters; new ones can be added by creating a new subclass of the *Filter* metaclass.

Finally, the *subject* of a rule identifies the programming language element to which the rule applies. The subject is always one of the elements defined in the *context*. In the original CCSL work [36], the subject defined the element on which an alert is raised in case a violation is identified. In this work the subject identifies the element that, if existing, will be selected as injection point.

Complex rules can be specified as a *CompositeRule*, which is essentially a connector to combine multiple rules using Boolean logic operators (and, or, etc.).

B. CCSL Specification Example

As an example of a CCSL specification, consider the following rule from the SEI CERT Coding Standard for Java [41], where the first part (before the ellipsis) is the actual rule, while the subsequent text is an explanation of the rationale.

"MET09-J: *Classes that define an equals() method must also define a hashCode() method. [. . .] The equals() method is used to determine logical equivalence between object instances. Consequently, the hashCode() method must return the same value for all equivalent objects. Failure to follow this contract is a common source of defects.*"

The corresponding CCSL specification is illustrated in Figure 3, as a diagram of metaclass instances (abstract syntax). Since the rule is composed of a simple statement, it is only necessary to create an instance of the *AtomicRule* metaclass (top left part of the figure). The subject of the rule is a Java class (*class* element, top right part of the figure) that contains a method named "equals".

However, only classes that define an "equals" method and do not define a "hashCode" method must be matched. This can be achieved by applying a filter. The *HasSuperClassFilter* recursively checks whether its target does *not* have (it is negated) a super class with certain characteristics, specified by the *filterContext*. In this case, the context of the filter includes an instance of *Method* with *name* set to "hashCode". Because the attribute *includesTarget* of the *HasSuperClassFilter* is set as true, the check is also performed on the *target* of the filter itself, that is, it is also checked if the current class does not contain a method named "hashCode".

It should be noted that, while an *Element* (an its subclasses) may in general have many different attributes, most of them have a minimum multiplicity of zero. This means that only the attributes that are needed for expressing the rule need to be specified, and all the others may be left empty. It should also be noted that the model in Figure 3 uses the abstract syntax of the language. Language engineering tools like Xtext [2] can be used to rapidly create textual editors for a DSL.

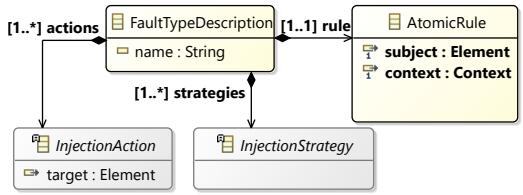

Fig. 4. Definition of a fault type description in EMF notation.

C. Fault Injection Extensions — Metamodel

In this work we have extended the CCSL to describe fault types, i.e., to define which elements will be modified in the source code (injection points) and which actions will be performed on the selected elements.

In order to specify fault types, we have created new metaclasses and reused some from CCSL. The root of a fault type specification is the newly introduced *FaultTypeDescription* metaclass, illustrated in Figure 4.

The main metaclasses involved in the specification of a fault type are described in the following.

FaultTypeDescription. This metaclass represents a fault to be injected in the source code. A *FaultTypeDescription* is composed of: i) a name; ii) a rule of type *AtomicRule*, which defines the elements of source code that will be targeted, i.e., the injection points; iii) actions to be performed on such elements, as a list of *InjectionAction*; and iv) injection strategies, as a list of *InjectionStrategy*.

AtomicRule. This is simply the *AtomicRule* metaclass imported from the original CCSL metamodel.

InjectionAction. This metaclass represents an action to be performed on its *target* element. The *target* element can be the injection point itself or part of it. The *InjectionAction* is an abstract metaclass, and it is extended by several concrete actions, each one containing its own implementation.

InjectionStrategy. While the *AtomicRule* metaclass defines a single rule which selects code elements as possible injection points, the *InjectionStrategy* defines a strategy to determine which of the possible injection points will be modified by the actions (e.g., only inject in 10% of the available injection points). The *InjectionStrategy* is an abstract metaclass, and it is extended by different concrete strategies, each one containing its own implementation.

To specify the faults listed in Table I the following subclasses of *InjectionAction* have been defined:

i) *DeleteAction* — deletes an element;
ii) *MoveScopeUp* — moves a statement up to the containing scope;
iii) *DeleteRandomStatementBlock* — deletes one random statement from a block;
iv) *ChangeLiteralValue* — changes the value of a literal value, e.g., the value of a literal number is incremented by 1;
v) *ReplaceArithmeticOperatorAction* — changes the operator of an arithmetic expression e.g., $a + b$ is changed to $a - b$; and
vi) *ReplaceVariableAccessAction* — replaces the access to a variable for another access of a variable of the same type, e.g., the invocation *foo(a)* can be replaced for *foo(b)*.

These actions allow to realize a wide range of different injections. It should be noted that additional actions can be added by creating new subclasses of *InjectionAction* in the metamodel.

In this paper we have defined and used only one subclass of *InjectionStrategy*: the *AllStrategy*, which simply selects all the candidate injection points. This strategy is also the one used by default when no strategy is provided in the fault type specification. As explained later, creating other strategies is only a programming exercise.

D. Fault Injection Extensions — Transformations

When CCSL is used to describe and check coding rules, only one file is generated from the specification: an OCL query to select elements with violations [36]. The OCL query is derived from the information contained in the rule specification, i.e., in an instance of the *Rule* metaclass.

Instead, the transformations that we implemented for this work take as input an instance of the *FaultTypeDescription* metaclass (Figure 4). As previously mentioned, two files are generated from such specification: i) an OCL query to select candidate injection points where the faults can be injected; and ii) a Java class file that concretely injects the fault into the selected injection points.

To generate the OCL query (`fault.ocl`) we basically reuse the transformations available in the CCSL framework, which are applied on the *rule* property of the *FaultTypeDescription* element. This property contains an instance of the *AtomicRule* CCSL metaclass, and thus can be used as input to the existing transformations.

The idea behind the Java class implementing the injector (`faultAction.java`) is simple. The class must contain a *doAction* method, which implements the action itself. The method receives the output of the OCL query as a parameter, thus basically obtaining a reference to the identified injection point in the model of the source code. Then, inside the *doAction* code, the model is altered as needed, using the APIs provided by EMF. This is possible because all the other toolsets (Eclipse OCL and MoDisco) are also based on EMF, and thus the output produced by the OCL query is an Ecore object (subclass of the *EObject* metaclass). To provide a unified access to the actions, the generated action must implement the *InjectionAction* interface, which contains the *doAction* method.

VI. Exemplary Execution of the Workflow

To concretely illustrate the overall workflow of Figure 1, we use here the fault *Missing Variable Assignment Using an Expression* (MVAE) from Table I as running example. We use circled numbers to recall the steps of the technical workflow.

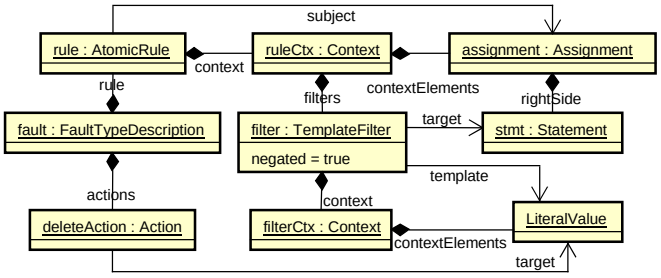

Fig. 5. Specification of the MVAE fault type using our language.

A. Specification

The first step ① is to create the structured specification of the fault to be injected using our DSL. We need therefore to identify i) the injection points, and ii) the actions to be performed. In this case the possible injection points are *all the variable assigments using an expression*, while the action is to *delete the assignment*.

The specification for the MVAE fault type is illustrated in Figure 5. The root of the specification is an instance of *FaultTypeDescription*, having an *AtomicRule* as the *rule* attribute. The *context* of the rule is defined as an *Assignment* where its *rightSide* is any statement (*Statement* is an abstract metaclass in CCSL). The *subject* of the rule is the assignment itself.

To select only assignments whose right side is an expression we use a *TemplateFilter*. This kind of filter, available in CCSL, provides an example of what should be matched in the source code. In this case, the filter checks if the *stmt* instance (i.e., the right side of the assignment) is compatible with the *LiteralValue* metaclass. However, in this specification the filter is negated (the *negated* attribute is set to *true*), and therefore only instances that *do not* comply with the provided template are selected. This means exactly that only assignments whose right side is not a literal are returned.

The set of *actions* in the fault specification only contains a *DeleteAction* (*deleteAct* instance). The *target* of the action is the *stmt*, which is the right side of the *Assignment*. Therefore, the right side of the *Assignment* will be deleted.

It should also be noted that the instance of *FaultTypeDescription* does not contain a strategy. This means that the *AllStrategy* will be used by default, i.e., all the existing "variable assignments using an expression" in the source code will be selected as an injection point.

B. Generation

Now that we have the specification of the fault we can generate the injector ②. As mentioned before, this step generates two files: an OCL query (`MVAE.ocl`) and a Java class (`MVAEAction.java`).

The OCL query for the MVAE fault type is shown in Figure 6. While it is not easy to read, it should be noted that it is completely generated in automated way by combining the information in the fault type specification with low-level knowledge of the Java language (embedded in the transformation). In particular, it should be noted that the matching

```
Assignment.allInstances()->select(assignment: Assignment |
  let stmt: ASTNode = assignment.rightHandSide
  ->asOrderedSet()->closure(v: ASTNode |
    if (v.oclIsKindOf(ParenthesizedExpression)) then
      v.oclAsType(ParenthesizedExpression).expression
    else
      v
    endif
  )->last() in stmt <> null and
  (
    stmt.oclIsKindOf(Statement) or
    stmt.oclIsKindOf(Expression)
  ) and
  not(
    let stmtTemplate: ASTNode = stmt in
    (
      stmtTemplate.oclIsKindOf(StringLiteral) or
      stmtTemplate.oclIsKindOf(CharacterLiteral) or
      stmtTemplate.oclIsKindOf(NumberLiteral) or
      stmtTemplate.oclIsKindOf(BooleanLiteral) or
      stmtTemplate.oclIsKindOf(NullLiteral)
    )
  )
)
```

Fig. 6. Generated OCL query for the MVAE fault type.

```
public class MVAEAction implements InjectionAction {

  @Override
  public boolean doAction(ASTNode root) {
    ASTNode target = getTarget(root);
    if(target != null) {
      return new DeleteAction().doAction(target);
    }
    return false;
  }

  private ASTNode getTarget(ASTNode root) {
    if (root instanceof Assignment) {
      Assignment assignment = (Assignment) root;
      return assignment.getRightHandSide();
    }
    return null;
  }
}
```

Fig. 7. Generated Java injector for the MVAE fault type.

of the *LiteralValue* element in the specification results in the matching of five different metaclasses in the Java metamodel, each corresponding to the concept of "literal value" of one of the primitive datatypes.

The Java injector for the MVAE fault type is shown in Figure 7. The `MVAEAction` class follows the implementation pattern discussed in the previous section, with the *doAction* method receiving a parameter of type *ASTNode*, that is, any element of the Java metamodel. This node is then processed based on the specified actions. In this case an instance of the *DeleteAction* class is created and executed. This particular case also includes a private method *getTarget*, which is used to retrieve the *target* of the action when it is different from the *subject* of the rule.

Now that the injector for the MVAE fault has been generated, it can be stored and used in multiple Java projects. If the fault specification changes, of course the OCL and Java files need to be re-generated.

C. Injection

The first step to actually inject the fault is to extract a structured model of the code of the target project ③, using

MoDisco [11]. The output of this step is a file in XMI format, `project.xmi`, which conforms to the Java metamodel.

Then, all the candidate injection points in the project are identified ④. This is done by executing the OCL query `MVAE.ocl` on the model of the project obtained in the previous step (`project.xmi`). The results of this step is a list of elements of the Java metamodel, that is a list of *ASTNode* objects, which identify the possible injection points available in the software.

Once that we have the candidate injection points, we need to select which one will actually be modified ⑤. This is done by executing the specified strategy, taking as input the list of the candidate injection points. We do not detail this step in this paper, as we always select all the injection points. Implementing other strategies simply means iterating ofter the list of *ASTNode* elements obtained in the previous step, and selecting a subset of them.

After the selection, we need to actually inject the faults ⑥. This is done by iteratively invoking the *doAction* method of the `MVAEAction.java` class, by passing each time one injection point as parameter (in the form of an *ASTNode* element). This injects the fault in the model representing the project, and thus the file `project.xmi` now contains the project code with the injected faults.

Finally, the concrete Java source code needs to be generated from the model ⑦. This step is performed by executing an Acceleo code generation module included in the MoDisco framework. The modified project is now available as Java source code, and can be used as needed.

VII. Experimental Campaign and Results

In this section we report the experimental campaign we performed to evaluate the proposed approach. We assess if the proposed approach can inject a target fault model in real software projects. We first discuss the experiment setup, in terms of target software projects and faults that are injected, and then we discuss the results. All the source code, models, and transformations are publicly available at [37].

A. Experiment Setup

As target systems, we selected 6 public projects written in Java from GitHub. We selected two of the most "starred" projects with size less than 30MB for each of the following categories: Cryptography, Security, and Artificial Intelligence. We applied the size limit only for the sake of experiment manageability. The selected projects are listed in Table II, including their name, selected version, and a short description.

The objective of the experiment is to inject all the 13 fault types described in Table I using our methodology, thus verifying its applicability for real fault injection campaigns. We have translated the description of each fault type into a machine-readable structured model, similarly to what has been shown in Figure 5 for the MVAE fault type. Such specifications are the input of our workflow, as discussed before.

TABLE II
Software projects selected for the experimental campaign.

GitHub Repository	*Description*
cryptomator/cryptomator 1.5.0-beta2	Multi-platform transparent client-side encryption of files hosted in the cloud.
libgdx/gdx-ai 1.8.2	Artificial Intelligence framework for games based on libGDX.
microsoft/malmo 0.37.0	Platform for Artificial Intelligence experimentation and research built on top of Minecraft.
spring-security-oauth 2.4.0.RELEASE	Support for adding OAuth1(a) and OAuth2 features for Spring web applications.
google/tink v1.3.0-rc3	Multi-language, cross-platform, open source library that provides cryptographic APIs.
Netflix/zuul v2.1.6	Zuul is a gateway service that provides dynamic routing, monitoring, resiliency, security, and more.

TABLE III
Overview on how the selected faults have been specified as a *FaultTypeDescription* (please also refer to Figure 4).

Name	*Rule*	*Actions*
MFC	Find a method invocation (*mInv*) where its container is a block, or any control flow statement (it avoids the deletion of an invocation such like int a = foo())	Delete *mInv*
MIA	Find an if statement (*ifStmt*) that contains any then statement (*thenStmt*) and does not have the else part	Move *thenStmt* one scope up and delete *ifStmt*
MIEB	Find an if statement (*ifStmt*) with any then statement and any else statement (*elseStmt*)	Move *elseStmt* one scope up and delete *ifStmt*
MIFS	Find an if statement (*ifStmt*) that contains any then statement and does not have the else part	Delete *ifStmt*
MLAC MLOC	Find an operand (*op*) in an AND/OR expression (*exp*) where *exp* can be find by making a recursively search in an if condition expression	Delete *op*
MLPA	Find a block (*block*) of a method declaration	Delete one random statement from *block*
MVAE MVIV	Find a literal value (*lit*) where *lit* is being used as a value of an variable assignment/initialization	Delete *lit*
WAEP	Find an arithmetic expression (*exp*) where *exp* is in a parameter of a method invocation	Replace the arithmetic operator of *exp* to another one
WPFV	Find a variable access (*varAccess*) where *varAccess* is in a parameter of a method invocation	Replace the (*varAccess*) to another variable access of a variable with same type.
WVAV	Find a literal value (*lit*) where *lit* is being used as a value of an assignment	Change the value of *lit*

Because of space constraints we are not able to show all the fault type specifications. Table III gives an overview on how fault types have been specified using our DSL. For all the fault types we have used the *AllStrategy*, meaning that the faults will be injected in all the candidate injection points. Other strategies can be implemented, as discussed more in details in Section VIII. We have used this strategy since the objective of this experiment is to evaluate if our tool is able to inject faults in a real software. The purpose of the experiment is not to verify the efficacy of injecting a specific set of faults,

TABLE IV
NUMBER OF INJECTED FAULTS FOR EACH FAULT TYPE.

Operator	*Injection Points*	*Successfully Injected*	
		Number	*Percentage*
MFC	15,261	15,261	100.0%
MIA	3,806	3,806	100.0%
MIEB	997	997	100.0%
MIFS	3,806	3,806	100.0%
MLAC	583	583	100.0%
MLOC	350	350	100.0%
MLPA	10,037	5,775	57.5%
MVAE	5,905	5,905	100.0%
MVAV	861	861	100.0%
MVIV	2,331	2,329	99.9%
WAEP	655	655	100.0%
WPFV	15,747	8,252	52.4%
WVAV	724	724	100.0%
Total	61.063	49.304	80.7%

for which a more accurate analysis of the requirements of the involved systems would be needed.

B. Results

Table IV summarizes the results obtained by executing all the generated injectors on the subject systems. The column "Injection Points" reports the total number of candidate code elements selected by the generated OCL query, while the two columns under "Successfully Injected" report the number and proportion of faults that were actually injected.

Our tool was able to find a total of 61,063 injection points and it successfully injected 49,304 faults (80.7%). For most fault types, 100% of injections were successful; exceptions are the MLPA, MVIV, and WPFV faults. Such exceptions are due to limitations in the information processed by the OCL query.

Consider for example WPFV, which consists in changing a variable used as a parameter of a method invocation to another of the same type. In this case the injection point has been specified as any variable that is used in a method invocation, while the action is responsible to find a new variable as replacement (see Table III). However, the generated OCL query may select a variable for which there is no other variable of the same type in scope, thus being impossible for the Java injector to actually inject the fault. When situations like this occur, they are signalled by the *doAction* method returning *false*.

We have also recorded the time required for injecting each fault type in the subject systems, as illustrated in Figure 8. The graph reports the total time required, in milliseconds, to identify the injection points in the project and to inject all the faults. However, most of the time is spent in the identification of the injection points, while the injection time is significantly smaller (more than one order of magnitude). Considering the amount of injected faults, we believe the performance is reasonable. The time required for each fault type is highly related to two aspects: i) the total number of injection points available in the project, and ii) the complexity of the generated OCL query, which depends on the fault specification. For example, the injection of MFC (Missing Function Call) appears as the most time consuming in three of the six projects. The MFC specification is quite simple, and it generates a small OCL query. However, function calls are among the most common statements in the code. Therefore, it is plausible that the retrieval and then deletion of all the function calls in a source code takes longer than other faults.

On the other hand, WAEP is the most time consuming in the other three projects, although only 655 injection points were identified across the six systems. Such a behavior occurs because its specification uses the *ArithmeticExpression* metaclass. When using such metaclass, the generated OCL is larger due to specific checks that guarantee that an expression is actually an arithmetic expression and not, for example, a concatenation of strings.

VIII. DISCUSSION

We discuss here on threats to validity and on possible extensions of this work. These extensions constitute our planned future work.

A. Threats to Validity

Our experiments investigated the use of model driven techniques to automatically inject faults in Java source code. The results indicate that the approach is feasible, as we successfully injected about 50,000 faults in six different projects. We argue that there is heterogeneity among projects, since they are real projects developed by different teams and they belong to different domains, as presented in Section VII-A.

We selected 13 fault types from the literature, and we specified them using our DSL. To assess the proper behaviour of our tool, large samples of the altered projects files have been manually inspected for each fault type, and no deviations from the expected behavior have been identified. Also, in all the cases the toolchain always finished the injection process with no visible failure. This means that the structured model used as input of stage ⑦ was syntactically correct (conformed to its metamodel), otherwise a warning would have been raised.

Last, it should be observed that, in our examples, all the injected faults can activate i.e., it is not possible to enable only a subset of the faulty pieces of code during a specific run. In fact, this paper focuses on the definition of the methodology to specify fault libraries. The definition of an experiment controller and of an injector for tailored fault seeding are outside the scope of the paper but represent our future works, as discussed below.

B. Extensibility of the Approach

We recall that in our DSL a fault type is represented by the *FaultTypeDescription* metaclass, which is composed of a rule (the pattern to be searched), actions, and strategies. By refining the fault type specification, it is possible to target more specific faults in different ways.

Fist, it is possible to add more sophisticated conditions in the *rule* that identifies suitable injection points. Consider for example the specification of MVAE in Figure 5. This selects all

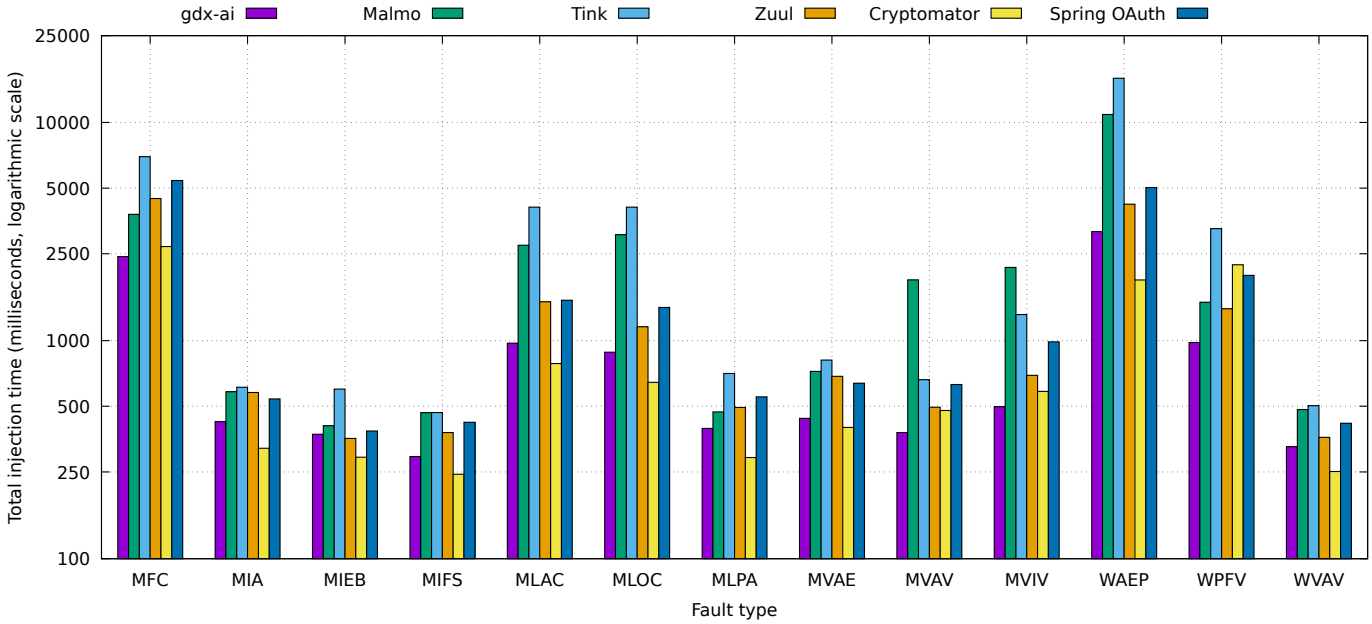

Fig. 8. Time required to inject each fault type in each target project. The value includes both the time to select injection points and the actual injection time.

the variable assignments that use an expression, and removes the assignment part. This of course results in a large number of injection points. However, this fault specification can be refined by adding more details on where these variable assignments should be searched for. As an example, we could inject faults only in methods of classes that extend an hypothetical *EncryptionAlgorithm* class. This can be done by simply adding a filter in the *ruleCtx* element in the specification of Figure 5, and more specifically, a *ContainerFilter*, which filters code elements based on their containing element.

After injection points are identified, in this work we only used the *AllStrategy*, injecting in all the available points. Creating other strategies is quite straightforward, as it is sufficient to extend the *InjectionStrategy* metaclass and provide the implementation of the strategy itself. This typically means to apply some filter to the list of candidate injection points identified by the OCL query. Example of strategies may be: i) select only the first n injection points; or ii) select n random injection points.

Another interesting option is to create fault types that inject known vulnerabilities. For example, we can inject violations to the previously mentioned MET09-J rule by selecting classes that define both the *equals* and the *hashCode* methods, and deleting the *hashCode* implementation. The specification to inject a violation of the MET09-J rule is illustrated in Figure 9; essentially, it has been derived by inverting the original CCSL specification of the MET09-J rule (Figure 3). We have used the proposed methodology to inject this fault type specification in the six selected projects. This resulted in 24 injection points, and the fault has been injected successfully in all of them.

Finally, the extensibility of the approach to other programming languages besides Java needs to be investigated. Most of the concepts in CCSL are common to many programming languages, e.g., class, method, variable, etc., and therefore

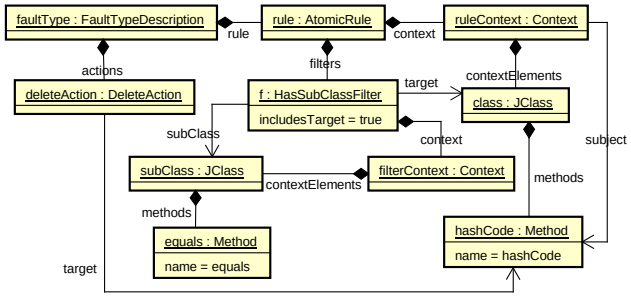

Fig. 9. Specification of the fault type introducing a violating the MET09-J rule from [41].

the fault specifications could be reused as they are. However, the transformations would need to be modified to adapt the injection process to the specificities of the language. This, as the rest of topics in this section, constitutes our future work.

IX. Conclusions

In this paper we proposed an approach to provide structured specifications of fault types, by applying model-driven engineering techniques. To the best of our knowledge, there is little work in this direction.

We have extended a language created to specify coding rules in a structured way, CCSL, to be able to also specify fault types. We then specified 13 common fault types and we applied them to six Java projects that belong to different domains. In general, our tool was able to find 61,061 injection points and to successfully inject 49,304 faults (80,7%), showing the promising use of model-driven engineering for fault injection.

As our language has been designed to be a high-level language where users do not need the knowledge of low-level artifacts like abstract syntax tree, it is possible to create custom faults and have them automatically injected for Java or other languages, as long as appropriate transformations are defined.

References

[1] Acceleo. https://www.eclipse.org/acceleo/. (Accessed August 17, 2020).

[2] Xtext: Language Engineering for Everyone! https://www.eclipse.org/Xtext/. (Accessed August 17, 2020).

[3] Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson. GOOFI: Generic object-oriented fault injection tool. In *2001 International Conference on Dependable Systems and Networks*, pages 83–88. IEEE, 2001.

[4] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[5] Iban Ayestaran, Carlos F Nicolas, Jon Perez, Asier Larrucea, and Peter Puschner. Modeling and simulated fault injection for time-triggered safety-critical embedded systems. In *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 180–187. IEEE, 2014.

[6] Paul Baker, Zhen Ru Dai, Jens Grabowski, Ina Schieferdecker, and Clay Williams. *Model-driven testing: Using the UML testing profile*. Springer Science & Business Media, 2007.

[7] Alexandre Bartel, Benoit Baudry, Freddy Munoz, Jacques Klein, Tejeddine Mouelhi, and Yves Le Traon. Model driven mutation applied to adaptative systems testing. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 408–413. IEEE, 2011.

[8] Tania Basso, Regina Moraes, Bruno P Sanches, and Mario Jino. An investigation of java faults operators derived from a field data study on java software faults. In *Workshop de Testes e Tolerância a Falhas*, pages 1–13, 2009.

[9] Simona Bernardi, José Merseguer, and Dorina C. Petriu. Dependability modeling and analysis of software systems specified with UML. *ACM Computing Surveys*, 45(1), 2012.

[10] Valentina Bonfiglio, Leonardo Montecchi, Francesco Rossi, Paolo Lollini, András Pataricza, and Andrea Bondavalli. Executable models to support automated software fmea. In *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*, pages 189–196. IEEE, 2015.

[11] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. MoDisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*, pages 173–174, 2010.

[12] Christian Buckl, Dominik Sojer, and Alois Knoll. Ftos: Model-driven development of fault-tolerant automation systems. In *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*, pages 1–8. IEEE, 2010.

[13] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, (4):171–188, 2005.

[14] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, and Roberto Natella. ProFIPy: Programmable Software Fault Injection as-a-Service. In *Proceedings of the 50th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2020)*, València, Spain, June 29-July 2 2020.

[15] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[16] Murial Daran and Pascale Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. *ACM SIGSOFT Software Engineering Notes*, 21(3):158–171, 1996.

[17] J. A. Duraes and H. S. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006.

[18] Pablo Gómez-Abajo, Esther Guerra, and Juan de Lara. Wodel: a domain-specific language for model mutation. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1968–1973, 2016.

[19] Pablo Gómez-Abajo, Esther Guerra, Juan de Lara, and Mercedes G Merayo. Mutation testing for dsls (tool demo). In *Proceedings of the 17th ACM SIGPLAN International Workshop on Domain-Specific Modeling*, pages 60–62, 2019.

[20] Rahul Gopinath, Carlos Jensen, and Alex Groce. Mutations: How close are they to real faults? In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 189–200. IEEE, 2014.

[21] William Hoarau, Sébastien Tixeuil, and Fabien Vauchelles. Fault injection in distributed java applications. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 7–pp. IEEE, 2006.

[22] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[23] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of mde in industry. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 471–480, May 2011.

[24] Tahar Jarboui, Jean Arlat, Yves Crouzet, Karama Kanoun, and Thomas Marteau. Analysis of the effects of real and injected software faults: Linux as a case study. In *Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing*, pages 51–58. IEEE, 2002.

[25] Henrique Madeira, Diamantino Costa, and Marco Vieira. On the emulation of software faults by software fault injection. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages

417–426. IEEE, 2000.

[26] Vince Molnár and István Majzik. Model checking-based software-fmea: Assessment of fault tolerance and error detection mechanisms. *Periodica Polytechnica Electrical Engineering and Computer Science*, 61(2):132–150, 2017.

[27] Regina Moraes, Hélene Waeselynck, and Jérémie Guiochet. UML-based modeling of robustness testing. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 168–175. IEEE, 2014.

[28] Aniello Napolitano, Gabriella Carrozza, Nuno Antunes, and João Duraes. Survey on Software Faults Injection in Java Applications. In *Innovative Technologies for Dependable OTS-Based Critical Systems*, pages 101–114. Springer, 2013.

[29] Roberto Natella, Domenico Cotroneo, Joao Duraes, and Henrique Madeira. Representativeness analysis of injected software faults in complex software. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 437–446. IEEE, 2010.

[30] Roberto Natella, Domenico Cotroneo, Joao A Duraes, and Henrique S Madeira. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering*, 39(1):80–96, 2012.

[31] Roberto Natella, Domenico Cotroneo, and Henrique S Madeira. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)*, 48(3):1–55, 2016.

[32] Nicolas Navet, Ivan Cibrario Bertolotti, and Tingting Hu. Software patterns for fault injection in cps engineering. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–6. IEEE, 2017.

[33] Object Management Group. Object Constraint Language. formal/2014-02-03. Version 2.4, February 2014.

[34] Object Management Group. Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM). formal/16-09-01. Version 1.4, September 2016.

[35] János Oláh and István Majzik. A model based framework for specifying and executing fault injection experiments. In *2009 Fourth International Conference on Dependability of Computer Systems*, pages 107–114. IEEE, 2009.

[36] Elder Rodrigues Jr. and Leonardo Montecchi. Towards a Structured Specification of Coding Conventions. In *Proceedings of the 24th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2019)*, pages 168–177, Kyoto, Japan, December 1-3 2019.

[37] Elder Rodrigues Jr., Leonardo Montecchi, and Andrea Ceccarelli. CCSL GitHub Repository. https://github.com/Elderjr/Coding-Conventions-Specification-Language (Accessed August 17, 2020).

[38] Bruno Pacheco Sanches, Tânia Basso, and Regina Moraes. J-SWFIT: A Java software fault injection tool. In *2011 5th Latin-American Symposium on Dependable Computing*, pages 106–115. IEEE, 2011.

[39] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[40] Adenilso da Silva Simão and José Carlos Maldonado. MuDeL: A language and a system for describing and generating mutants. *Journal of the Brazilian Computer Society*, 8(1):73–86, 2002.

[41] Software Engineering Institute – Carnegie Mellon University. SEI CERT Coding Standard. https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards/ (Accessed August 17, 2020).

[42] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, 2008.

[43] Tricentis. Software fail watch: 5th edition. White paper, avilable at https://www.tricentis.com/resources/software-fail-watch-5th-edition/ (Accessed August 17, 2020), 2017.

[44] Markus Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, January 2013.

[45] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2014.