Characterizing and Injecting Faults in Executable Models Specified with fUML

Guilherme W. Lopes
Institute of Computing
Universidade Estadual de Campinas
Campinas, SP, Brazil
g230253@dac.unicamp.br

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

***Abstract*—Fault Injection (FI) is a well-known system verification technique, in which faults are artificially introduced into a system, to assess its behavior in exceptional conditions. FI can be applied at different levels, including physical, hardware and software. FI has also been applied at model level, although the amount of work in this direction is limited. However, the importance of models in the development of complex systems is growing, pushing towards model-level verification and simulations. The Foundational UML (fUML) is a specific subset of UML, released as an OMG standard, that has a precise execution semantics and can therefore be executed. In this paper we investigate the application of FI to executable models specified with fUML. We first characterize the kind of fault that may be applied to fUML models, and then we realize an injection mechanisms based on automated model transformation. We apply the methodology to a simple example in the space domain, adapted from a real satellite system. We believe this is an important first step for the adoption of FI techniques on fUML models, for an early detection of design flaws. We conclude discussing some challenges and directions for future work.**

***Index Terms*—fault injection, fUML, executable models, model-driven, fault model**

I. Introduction

Fault Injection (FI) [1] is a well-known technique for the assurance of critical systems. In complex systems, incorrect behavior may be caused by very specific conditions that are difficult to reproduce in the assessment phase. The idea of FI is to artificially introduce (*inject*) faults into the system, in order to assess the effectiveness of implemented countermeasures.

In this perspective, FI is a way to accelerate the occurrence of faults during the verification process of a system [2]. Different FI techniques have been used for different purposes, including validation of fault tolerance mechanisms implemented in specific systems, dependability benchmarking of different systems, and fault forecasting [3].

FI can be applied at different levels, including physical, hardware, and software. Some works have also applied FI at model level, although the amount of work in this direction is limited. On the other hand, the importance of models in the development of complex systems is growing. In the Cyber-Physical Systems (CPS) domain, the *digital twins* [4] trend clearly pushes towards increased importance of model-level verification and simulations.

Model-Driven Engineering (MDE) [5] has contributed to the integration between modeling and subsequent development phases, through formalization of models, automation of derivation tasks and constraint-checking mechanisms. MDE was one of the driving forces behind the advent of *executable models*, that is, models that can be executed like real source code. In particular, the Foundational UML (fUML) is a specific subset of UML having a precise execution semantics and can therefore be executed. fUML has been released as a standard by the Object Management Group (OMG) in 2011 [6].

In this paper we investigate the application of FI to executable models specified with fUML. We first characterize the kind of fault that may be applied to fUML models, and then we realize an injection mechanisms based on automated model transformation. We apply the methodology to a simple example in the space domain, adapted from a real satellite system. To the best of our knowledge, fault injection in fUML models has not been investigated yet.

The rest of the paper is organized as follows. Section II introduces the background to our work, while Section III provides a brief review on executable UML models. In Section IV we provide an overview of the proposed approach, whose steps are then detailed in the subsequent sections: in Section V we describe the characterization of faults for fUML models, in Section VI we describe how faults are annotated into existing models, and then in Section VII we describe how injection is performed. A small example is discussed in Section VIII, while in Section IX we discuss the related work. Finally, conclusions are drawn in Section X.

II. Background

A. Fault Injection

An accurate discussion of the different fault injection techniques can be found in the work from Arlat et al. [7]. FI can be performed at different levels in the system. A broad categorization of techniques is between *physical* techniques, in which faults are physically injected in the system, and *simulation-based* techniques, in which models or simulators are used.

One of the oldest physical FI techniques consists in bombarding the system with magnetic interference or ions, to reproduce the effect of radiation encountered in space (e.g.,

gamma rays) [7]. Hardware-Implemented Fault Injection (HIFI or HWIFI) is realized by directly altering the state of integrated circuits, for example through pin-level ports used for debugging or by "bridging" (i.e., shorting) pins together. These techniques emulate the occurrence of faults by applying their effects to the hardware. Software-Implemented Fault Injection (SWIFI) consists in emulating the effects of physical faults by injecting modifications in the software. SWIFI can be applied at pre-runtime, by altering the code or data of the software before execution, or at runtime, by injecting erroneous data or altering the control flow during software execution [7].

More recently, Model-Implemented Fault Injection (MIFI) techniques have emerged [8], following the progressive adoption of models in the development of complex systems. MIFI consists in a family of techniques in which FI mechanisms are developed as rules for the modification of some kind of model, e.g., hardware models, when no physical prototypes are available yet; software models, to validate the software design; or system models, for example to understand the effect of faults in sensors or in communication.

The effectiveness of FI is limited by the precision achieved in the emulation of faults. That is, the faults that are injected must be representative of real faults that may be activated during system execution. Therefore, one of the main aspects of FI is selecting the kinds of faults of interest ("what to inject"), and where they could manifest themselves ("where to inject") [9]. The *fault model* defines, for a certain system and context, which are the faults that are considered possible and their immediate effects. For example, in a networked system the fault model may include assumptions on loss of messages, partitioning of the network, ordering of messages, or hang of nodes [10].

B. Model-Driven Engineering

MDE [5] is a methodology what advocates the systematic use of models as primary artifacts throughout the engineering lifecycle. Information should be described and managed at the most abstract level as possible, and concrete artifacts are generated from this primary information. MDE techniques combine: i) Domain Specific Languages (DSLs) [11], which formalize the information relevant for a certain domain; and ii) model-transformations and code generators [12], 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.

The MDE panorama has been strongly influenced by he Object Management Group (OMG), an international consortium that develops and maintains a number of standards related to MDE. The most popular one is probably the UML standard [14], which is widely used in the industry. Besides, the OMG maintains standards for many different MDE tasks, including metamodeling [15] and model transformation [16]. Among those, the fUML standard [6] addresses executable models.

III. Executable UML Models

With the introduction of MDE concepts, the UML gained a rigorous definition, in terms of its metamodel. However, it was still a semi-formal definition, as many aspects in the UML standard are left to interpretation. Approaches to give a formal execution semantics to UML models exist since long time. For example, the UML/P adaptation by Rumpe [17] or the "Executable UML" (xUML/xtUML) [18] have been devised more than a decade ago. Furthermore, the literature features multiple works in which parts of UML are precisely defined for a certain domain or task.

A. Foundational UML

Because the different attempts at executing UML models were substantially incompatible, the OMG developed the "Semantics of a Foundational Subset for Executable UML Models", in short *Foundational UML*, or simply fUML. The fUML standard [6] defines a precise semantics for a precise subset of UML, thus enabling its execution.

fUML uses only a subset of the UML metamodel, limiting its scope to the *Class Diagram* and the *Activity Diagram*. Even considering only these diagrams, some elements are not allowed in fUML, for example *OpaqueExpression* elements and all the elements involving time and time intervals are excluded. We provide here a brief review of the main elements of the fUML metamodel.

Execution is based on UML *Behavior*; in general, a *Behavior* can be executed by direct invocation, or by the creation of an active object having such behavior [6]. In particular, fUML is centered around *Activity* elements, a kind of *Behavior* described by a diagram composed of nodes and edges, more precisely *ActivityNode* and *ActivityEdge* elements (Figure 1). Although they are also composed of nodes and edges, Activity Diagrams should not be confused with state machines, which have a completely different syntax and semantics.

Three kinds of nodes are allowed in an *Activity*. An *ExeuctableNode* (basically, an *Action*) represents the execution of some other behavior, while a *ControlNode* may alter the control flow of actions within the activity. Actions and activities may receive objects as parameters and emit objects as results; each parameter is specified by an *ObjectNode* element (basically, a *Pin*), owned by the action (see Figure 2).

Edges can be of two different kinds. *ControlFlow* edges connect *ExecutableNode* and *ControlNode* elements, basically defining the execution order of the actions in the diagram

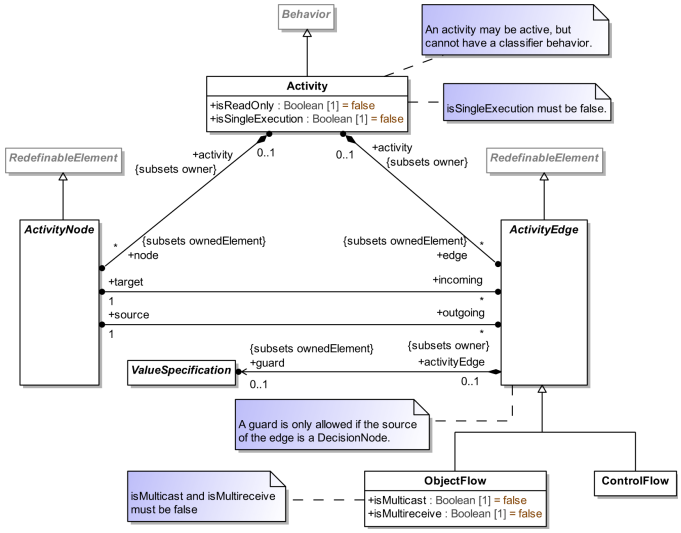

Figure 1: fUML Metamodel — *Activity* [6].

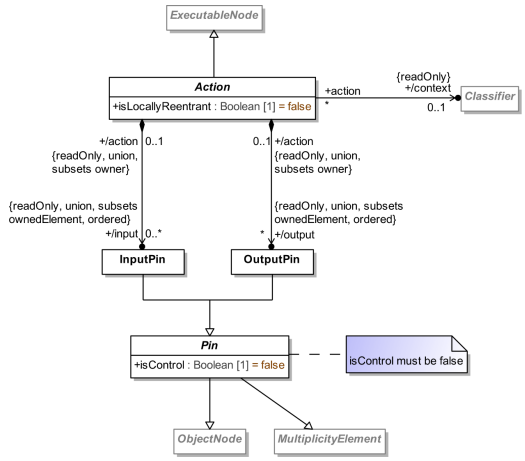

Figure 2: fUML Metamodel — *Action* [6].

(i.e., the control flow). *ObjectFlow* edges connect *ObjectNode* elements, describing how data is passed through the different actions.

B. Tool Support

To date, a limited number of tools support the execution of fUML models. A reference implementation of a fUML execution platform is available at [19]. In our experience, the tool is very strict on the format of the input model, and cumbersome manual modifications of the XMI file are required on models edited with common graphical editors.

The Moliz tool [20] was probably one of the first tools capable of executing regular UML models produced by the Papyrus diagram editor (provided of course that they are fUML-compliant). Later, the Papyrus "Moka" plugin integrated model execution in the main Eclipse Papyrus project. Currently, Moka supports various execution semantics, including fUML. Moka also supports the PSCS (Precise Semantics of UML Composite Structure) standard [21], a more recent standard

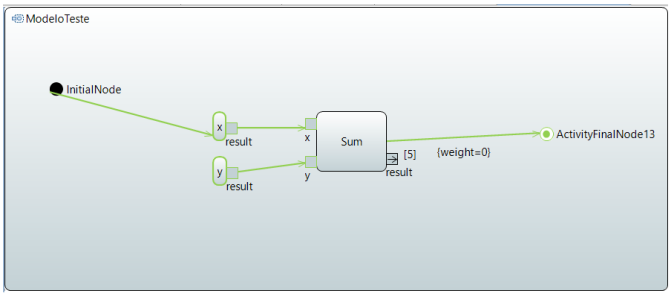

Figure 3: Example of a fUML-compliant model, and its execution using Moka.

that extends execution semantics to UML Composite Structure Diagrams.

A review and comparison of the mentioned fUML tools can be found in [22]. In this work we use Moka as execution framework, for being integrated in one of the most popular open source diagram editors. Figure 3 shows an example fUML model being executed using Moka. During the execution, elements that are being executed are highlighted in green, allowing the user to visually follow the execution steps.

IV. Fault Injection in fUML: Overview

In this section we provide an overview of the methodology proposed in this paper. Figure 4 summarizes the adopted workflow, highlighting the main steps and the tools adopted in each of them.

The first step ① involves understanding how the system of interest can be modeled using fUML elements. As described in Section III, fUML mainly includes elements from the Activity and Class diagrams of UML, with some additional restrictions to enable execution. From such analysis, the base system model is created, which describes the nominal behavior of the system under analysis. It should be noted that the model may be a design model, to be used in the subsequent development phases, or it may just serve as a verification model, to verify the behavior of an existing system.

In the second step ②, faults to be injected are attached to the model, based on which system requirements need to be tested. This step is based on the characterization of faults that may be injected into an fUML model, which is one of the main contributions of this paper. The characterization, described in Section V, has been defined based on previous work on the literature, adapting existing fault types to the fUML domain.

The third step ③ consists in actually injecting the faults in the executable model of the system. In our proposal, this step is performed automatically by model transformation. The transformation receives as input the nominal fUML model of the system and a specification of the fault(s) to be injected, and it produces as output a modified fUML model affected by the fault(s). The design and implementation of the transformation is described in Section VII.

Finally ④, the modified model is executed, and its output is analyzed to determine the effects of the injected faults. This

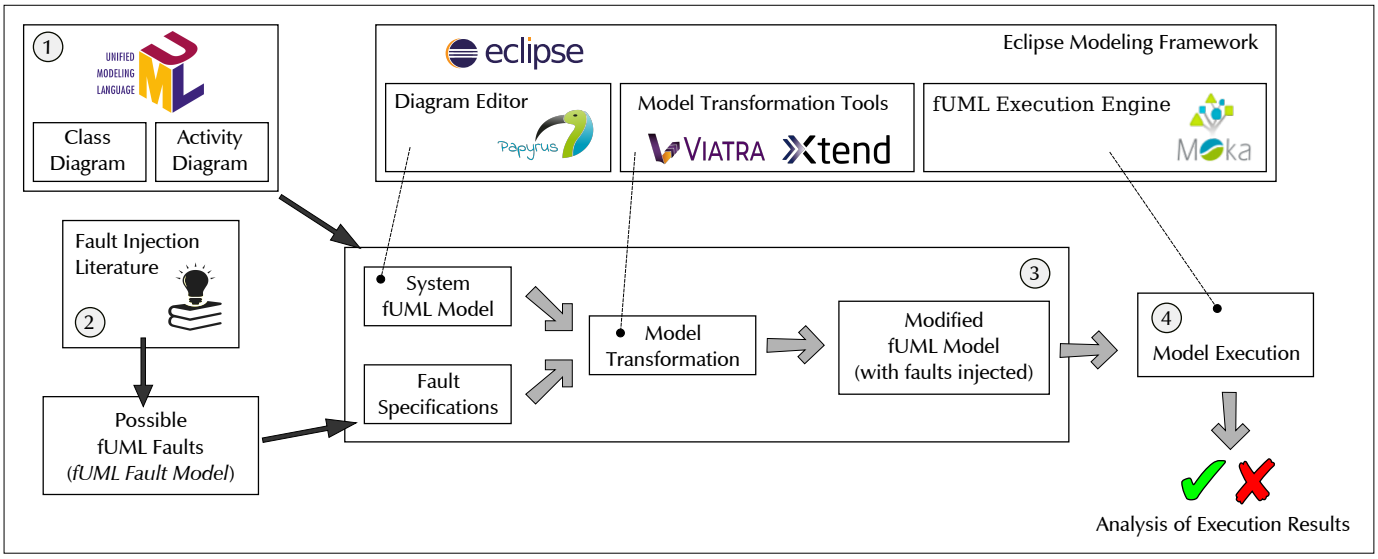

Figure 4: Overview of the approach proposed in this paper.

kind of analysis is not trivial, and it is related to the oracle problem in testing [23], that is, knowing the expected output of the system. This aspect is out of the scope of this paper, and we assume that the expected output of the system is known, for example, obtained by a golden run of the nominal model.

Steps ① and ④ are based on existing tools, namely Eclipse Papyrus and Moka. Steps ② and ③ are the main contribution of this paper, and they are described in the following.

V. Characterization of fUML Faults

In this section we define which faults can be injected in fUML models according to our approach, and how they are specified.

A. Fault Model

We base our fault model on established work in the literature on the fault injection topic. In particular, we took as reference the work of Duraes and Madeira [24], later adopted also in [9]. The experimental work in [24] identified a wide list of fault types encountered in real software; each of these fault types was described by an acronym and a short description, effectively forming a taxonomy of software faults. The work also identified a set of most common faults occurring in real software, which is reproduced in Table I.

The characterization of faults that are relevant for fUML models has been identified in two steps. First, the possible modifications to fUML models have been analyzed, going through the main metaclasses involved in a typical fUML model. Then, the identified faults have been mapped to the taxonomy in [24]. The result of this analysis is reported in Table II. We note that we focused on pre-runtime faults only, that is, faults that are injected in the model before its execution. This choice is due to the injection method that we adopt, model transformation, which works on the structure of the model itself. Runtime injection is possible, e.g., on a running Moka execution, but it is out of the scope of this paper.

Table I: Most common software faults occurring in the field, according to the characterization proposed by Duraes e Madeira [24], [9].

Type	*Description*
MFC	Missing Function Call
MVIV	Missing Variable Initialization Using a Value
MVAV	Missing Variable Assignment using a Value
MVAE	Missing Variable Assignment Using a Expression
MIA	Missing IF construct Around Statements
MIFS	Missing IF construct plus Statements
MIEB	Missing IF construct plus Statements plus ELSE Before Statements
MLC	Missing AND/OR clause in branch condition
MLPA	Missing small and localized part of the algorithm
WVAV	Wrong Value Assigned to Variable
WPFV	Wrong Variable used in Parameter of Function Call
WAEP	Wrong Arithmetic Expression in Parameter of Function Call

The fault types in Table II are grouped by the kind of diagram to which they apply (Class or Activity), and by the metaclass they affect. Each fault is mapped to one of the fault types identified in [24]. Most of them have been mapped to the common fault types listed in Table I; however, we found that five additional fault types are relevant for defining a fault model for fUML, namely WBC1 (*Wrong Branch Construct – Goto Instead Break*), WSUT (*Wrong Datatype or Conversion Used*), WVIV (*Wrong Value used in Variable Initialization*), WALD (*Wrong Algorithm – Small Sparse Modifications*), and WALR (*Wrong Algorithm – Code Was Misplaced*). For example, modifying the type of an input (or output) pin of an activity, is clearly an instance of the WSUT fault type.

Table II also lists a *Representativeness* column, which is described in the following subsection.

B. Representativeness

As a first step towards understanding representativeness of fUML faults, we classified them based on how they are likely

Table II: Type of faults that may affect an fUML model and their relation with the fault characterization defined in [24]. Acronyms of faults are listed in Table I, except for: *WBC1: Wrong Branch Construct – Goto Instead Break*, *WSUT: Wrong Datatype or Conversion Used*; *WVIV: Wrong Value used in Variable Initialization*; *WALD: Wrong Algorithm – Small Sparse Modifications*; and *WALR: Wrong Algorithm – Code Was Misplaced.*

Diagram	*Metaclass*	*Fault Description*	*Mapping to [24]*	*Representativeness*
Class	Property	Modify the type of the property	WVAV	Low
	Class	Modify the connections between classes	WBC1	Medium
	Operation	Modify the variable used in the parameters of the operation	WPFV	High
Activity	ControlFlow	Substitute a ControlFlow element with an ObjectFlow element	WBC1	Low
	ObjectFlow	Substitute an ObjectFlow element with a ControlFlow element	WBC1	Low
	ControlFlow	Modify the direction of the connection between two ControlFlow nodes	WBC1	Low
	Guard	Modify the guard expression	WVAV	High
	Object	Remove the reference to the object	MVIV	High
	DecisionNode	Modify the decisions	MIFS	High
	ActivityNode	Remove the node	MFC	Medium
	ValueSpecificationAction	Modify the value produced by the action	WVIV	High
	ValueSpecificationAction	Modify the type of the object produced by the action	WSUT	Medium
	CallBehaviorAction	Modify the order of parameters that are connected to the action	WALD	Medium
	CallBehaviorAction	Invert the order of elements in a comparison action	WALR	High
	ForkNode	Remove the fork node	MLPA	Medium
	ValueSpecificationAction	Modify the type of the action specification	WSUT	High
	Pin	Remover an input or an output pin	MLPA	Low
	Pin	Modify the type of the pin	WSUT	Low

to actually appear as a residual fault in a fUML model of a system. That is, how much they are *representative* of real faults that may remain unnoticed in real fUML models. This is of course a first classification effort, which would require a more in-depth analysis, and, possibly, field studies on fUML models of real systems.

The classification of faults based on their representativeness is based on a simple heuristic: "how easy is to detect the fault in the model", which basically defines the chance that the fault propagates, undetected, to the next development phases. Based on this concept we define three representativeness classes:

- *Low*. The fault can be detected easily and consistently. Typically, this means that the fault can be detected automatically by some simple constraint-checking tool.
- *Medium*. The fault can be detected when the model is analyzed with care. Typically, some important part of the model is missing or modified with respect to the intended functionality. However, the fault cannot be detected automatically.
- *High*. The fault is difficult to detect because it makes some subtle modification to the model. The model needs to be thoroughly analyzed or tested in order to identify the fault. For example, the modification of a branch condition falls in this category.

The result of classification is reported in the last column of Table II. For example, substituting a *ControlFlow* element with an *ObjectFlow* element has been considered to have *low* representativeness. In fact, even though the model would still execute (likely resulting in wrong behavior), the fault can be easily identified by the (f)UML validator provided with Eclipse Papyrus. Conversely, modifying a guard expression has *high* representativeness, because it can only be identified with thorough testing of the model.

Figure 5: A screenshot of the MoDisco Model Browser showing an UML Comment and its properties.

VI. Annotation of Fault Specifications

After having identified the fault types relevant to fUML models, the next step is to define how a modeler should specify them in order to trigger the injection.

A. Attaching Fault Specifications to UML Comments

One of the simplest ways to extend an UML model with custom information is by using comments. Comments are instances of the UML *Comment* metaclass [14], which accept arbitrary text in their *body* property. Furthermore, a comment can be attached to one or more elements of the diagram, through the *annotatedElements* association. Therefore, we use UML comments for specifying the fUML element(s) on which the fault should be injected, and the details of the fault.

Figure 5 shows the structure of an UML Comment, visualized using the MoDisco Model Browser [25]. In this example, the fault applies to a *DecisionNode* object of *Activity* "Activity1", which are referenced into the *annotatedElement* property. Note that, when adding a *Comment* to an Activity

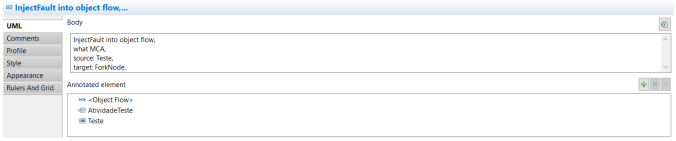

Figure 6: Annotating the fUML models with a fault specification using Eclipse Papryrus.

Diagram, the main *Activity* element is always added as an *annotatedElement* of the comment. Figure 6 shows how the annotation is performed by an end user, using the Eclipse Papyrus editor.

For the transformation to work, and thus for the fault to be correctly injected, two conditions must hold: i) the *body* of the comment must conform to the specification language that we devised specifically for this work, and ii) the fault type (e.g., WALD) must be compatible with the object referenced in the *annotatedElement* property, according to the mapping in Table II. The specification language is detailed in the next subsection.

B. Notation for the Specification of Injections

To define the details of the fault to be injected, we defined a small DSL, based on the following keywords:

- `InsertFault` — Identifies the begin of a fault specification.
- `into` — Identifies the element in the model where the fault will be injected (metaclass).
- `wherepin` — This keyword is used only in case of faults that are injected on pins of diagram elements. It identifies the pin, among those owned by the main object.
- `what` — Identifies the kind of fault to be injected, according to the acronyms in Table II.
- `source` — This keyword is used for faults whose application involves connectors, to identify the source of the connection.
- `target` — This keyword is used for faults whose application involves connectors, to identify the target of the connection.

For simplicity we assume that elements in the fUML model have unique names. This is typically the case, as duplicate identifiers are considered an error by editors like Papyrus. In any case, enforcing unique identifiers in a model is trivial, as it only requires appending a counter to element names.

Two examples of fault specifications using the proposed notation are listed in the following.

```
InjectFault into Pin,
wherepin "result",
what MLPA
```

```
InjectFault into ObjectFlow,
what WBC1,
source = object.target,
target = object.source
```

The first specification injects a MLPA fault (*"Missing small and localized part of the algorithm"*) into the pin *result* of the attached node, effectively removing the pin and all its connections. The second specification injects a WBC1 fault (*"Wrong Branch Construct – Goto Instead Break"*) into an ObjectFlow element, inverting the connection. In fact, the specification swaps the target and the source of the ControlFlow object, setting them to *object.source* and *object.target*, respectively.

The complete grammar of our language, in BNF (Backus-Naur Form), is defined as follows:

$$
\begin{array}{rcl}
\langle \text{spec} \rangle & \models & \text{InsertFault into } \langle \text{pattern} \rangle, \text{ what } \langle \text{fault} \rangle, \\
\langle \text{pattern} \rangle & \models & \langle \text{objname} \rangle, \mid \langle \text{objname} \rangle \text{ wherepin } \langle \text{pin} \rangle, \\
\langle \text{objname} \rangle & \models & \langle \text{instance} \rangle \mid \langle \text{metaclass} \rangle \\
\langle \text{fault} \rangle & \models & \langle \text{simplefault} \rangle \mid \langle \text{complexfault} \rangle \ \langle \text{arc} \rangle \\
\langle \text{simplefault} \rangle & \models & \text{WVAV} \mid \text{WPFV} \mid \text{MIFS} \mid \text{MFC} \\
 & & \mid \text{WSUT} \mid \text{WVAV} \mid \text{MLPA} \\
\langle \text{complexfault} \rangle & \models & \text{WBC1} \mid \text{MVIV} \mid \text{WALD} \\
\langle \text{arc} \rangle & \models & \text{source} = \langle \text{objname} \rangle, \ \text{target} = \langle \text{objname} \rangle \\
\langle \text{instance} \rangle & \models & \textit{String identifier of an element of the model} \\
\langle \text{metaclass} \rangle & \models & \textit{String identifier of a fUML metaclass}
\end{array}
$$

The language supports the specification of 10 fault types. Of these, 7 are considered "simple" faults, because their specification only involves the name of the fault and the object to which it is applied, without the need of additional information from the user. Conversely, the other 3 fault types are considered "complex" faults, because their injection requires additional information in the form of the source and target of a fUML connector.

It should also be noted that we provide the option to specify references to fUML elements either by their identifier (⟨instance⟩ rule), or by their type only (⟨metaclass⟩ rule). In case every element of the model has an identifier, the first option permits to precisely identify the intended element. However, this may not always be the case: often some elements are left without identifier, or the identifier is hidden from the user. This is particularly common for connectors in Activity diagrams (i.e., instances of the *ObjectFlow* or *ControlFlow* metaclasses). For this reason, we also support specifying the the metaclass name only, leaving to the transformation the task to identify the correct element, starting from the *ownedElements* property of the *Comment*, and the context of the model.

Figure 7 show an example using both notations. The *source* element is specified by its identifier ("Teste"), while the *target* element is specified by its metaclass only ("ForkNode"). In this case, the injection will create an *ObjectFlow* element connecting the *Teste* element and the element of type *ForkNode* close to it, on the left.

VII. Transformation Implementation

In this section we describe the implementation of the model transformation that actually perform the injection of the fault. The transformation takes as input an fUML model extended

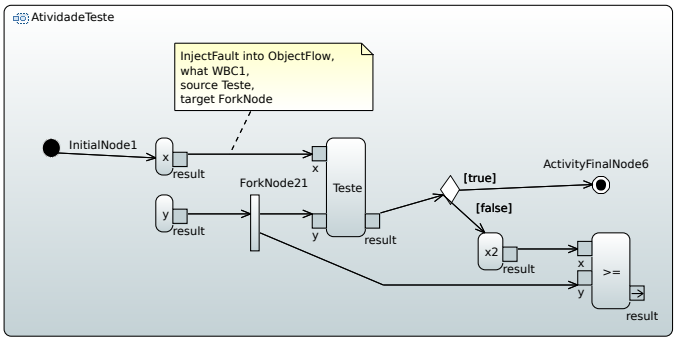

Figure 7: Specification of a fault to be injected, adding the fault type information in the body of a UML Comment.

Table III: Steps for the injection of a MFC fault on a *Node* element.

Metaclass	*Fault Type*	*Fault Description*
ActivityNode	MFC	Remove the node
Identify the node that will be removed For each flow incoming to the node: Identify the source and target properties of the incoming flow Select one of the flow exiting the node Identify the source and target properties of the outgoing flow Set target of the incoming flow to target of the outgoing flow Remove the outgoing flow Remove the node		

with fault specifications, and generates a modified, faulty, fUML model.

A. Transformation Algorithm

The input to the transformation is an UML model that contains at least an Activity Diagram. Besides that, the model must comply with fUML, i.e., it shall include only UML elements from the fUML subset. If these conditions are satisfied, the transformation shall trigger an injection for each UML *Comment* whose *body* conforms to the grammar in Section VI-B. Also, as explained before, the specified fault type must be compatible elements referenced by the comment.

Each fault type described in Section V has been analyzed and the steps required for its injection have been defined using pseudocode. The injections steps have been devised to ensure that a correct model is generated, whenever possible. With *correct* here we mean a model that conforms to the fUML metamodel.

Table IV: Steps for the injection of a WALD fault on a *CallBehaviorAction* element.

Metaclass	*Fault Type*	*Fault Description*
CallBehaviorAction	WALD	Modify the order of parameters that are connected to the action
Identify the action that will be modified Identify the incoming flows of ObjectFlow kind For each ObjectFlow incoming to the node Set the target property to a different pin of the same action		

```
//Call Behavior and WALD
pattern waldSpecification(act : Activity, c : Comment,
                          anode : CallBehaviorAction,
                          str : java String)
{
    Comment.annotatedElement(c, anode);
    Comment.body(c,str);
    Activity.ownedNode(act, anode);
    check (
        str.startsWith("InjectFault into behavior")
        && str.indexOf("WALD") > 0
    );
}
```

Figure 8: Viatra pattern for identifying WALD specifications that apply to *CallBehaviorAction* elements.

Table III and Table IV show the sequence of steps needed to inject two different faults types: MFC, a simple fault, and WALD, a complex fault. In more details, Table III lists the steps required to inject a fault of type MFC (Missing Function Call) on *Node* elements of a fUML diagram. Since this is a simple fault, no additional paramters are needed. Table IV details the steps required to inject a fault of type WALD (Wrong Algorithm – Small Sparse Modifications) on elements of type *CallBehaviorAction*. This fault accepts parameters from the user, namely the new *source* and *target* elements that will be used in the modified model. However, if they are not specified, the algorithm will select a suitable element from those available in the input model.

B. Injector Implementation

The injection is performed by a transformation that generates a modified model. The transformation is implemented with a combination of the Viatra [26] and the Xtend [27] tools. This is a common configuration for model transformation, in which Viatra is used to find the relevant elements in the model, and Xtend is used to actually perform the modifications.

Also in our case, Viatra is then responsible for identifying the annotations of specific fault types in the model. For each fault type in Table II we created a Viatra *pattern* to retrieve the model elemens annotataed with that fault. An example pattern for the WALD fault on *CallBehaviorAction* elements is shown in Figure 8.

A Viatra pattern specifies as parameters the kind of objects it works on. In this case, the WALD pattern applies to: i) an *Activity* (the main activity of the diagram), ii) a *Comment* (the fault annotation), iii) a *CallBehaviorAction* (the target of injection), and iv) a *String* (the fault specification itself).

The pattern identifies a WALD annotation if: i) the *Comment* is connected to the *CallBehaviorAction*, ii) the *Comment* has the *String* as its *body*, iii) the *Activity* contains the *Comment*, and iv) the *String* is a WALD specification. When the elements satisfying the pattern conditions are identified, they are passed as a tuple to the Xtend transformation code, which applies the required manipulation steps specific for each fault (e.g., Table IV for WALD).

The transformation always tries to produce a correct model, by taking into account the context around the injection point. However, this is not always possible (e.g., removing an *Action* and not having a compatible one to connect the dangling flows). This is however a common problem in fault injection, and it is not specific to our approach.

VIII. Example Application

In this section we show how our approach can be applied in practice, using a nanosatellite system as running example.

A. System Overview

The NanosatC-BR2 is a nanosatellite developed by the National Institute of Space Research of Brazil (*Instituto Nacional de Pesquisas Espaciais*, INPE) and other partners, following the CubeSat standard. The objective of the satellite is to study the Earth's ionosphere and magnetosphere; in particular, the disturbances in the region of the South Atlantic Anomaly (SAA), which cause negative effects in telecommunications and localization services like GPS [28], [29].

The NanosatC-BR2 adopts a 2U CubeSat platform, meaning that it is composed of two basic units (2U) of the platform, connected together. The satellite carries the On-Board Computer (OBC) and a number of payloads, that is, software that provides some functionality to the satellite, besides the basic operations provided by the OBC. Among the payloads carried by the NanosatC-BR2, the *Langmuir Probe* (LP) is a scientific component used to measure electron numerical density and other properties of plasma. Another payload is the *Attitude Determination System* (ADS), which calculates the orientation of the satellite with respect to three axes, based on magnetic measurements collected from embedded sensors [30].

During the development of the system, modeling has been extensively used to support the specification of requirements and its later verification. However, software models were primarily defined using the UPPAAL language [30].

B. fUML Model of the Langmuir Probe

Based on the report in [30] and on the requirements of the system, we created an fUML model of the LP component, to use it as base model for showing the applicability of our approach.

The LP exposes a single behavior: the acquisition of data from the probe itself and its transmission to the OBC. The transmission of data to the OBC occurs in chunks: the probe collects a certain amount of data, filling its internal buffer, and the buffer is then transmitted to the OBC.

We first defined the LP component as a *Class*, with its attributes and behaviors (Figure 9). This step is necessary in fUML when we want to model behavior that operates on the internal state of a component, because UML *Activity* elements do not hold state.

The Activity Diagram in Figure 10 represents the main behavior of the LP component. The probe is first initialized (*Initialize* action) and then the *GatherData* action is executed. *GatherData* returns *true* if there is still data to be collected,

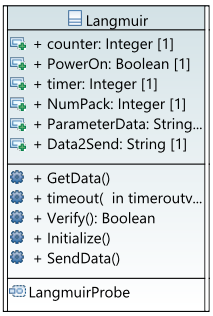

Figure 9: Class Diagram to specify the attributes and behaviors of the LP component.

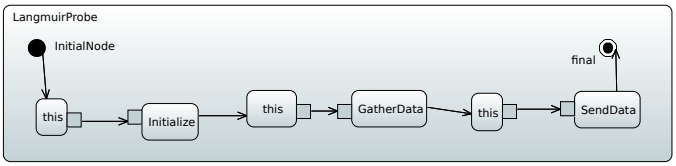

Figure 10: Model of the main behavior of the LP payload.

and in this case it is executed again. After all data has been collected, the data is sent to the OBC (*SendData* action) and then the activity terminates.

C. Fault Injection and Results

When needed, fUML actions can be refined by more detailed diagrams in a top-down approach. This means that the internal specification of an action can be detailed by a separate Activity Diagram describing the implementation of the action itself. While it is not possible to detail the entire model in this paper, we focus here on the *GetDataPack* activity, which is called inside the *GatherData* action of Figure 10.

The *GetDataPack* activity (Figure 11) represents the collection of a single data element from the sensor, and its storage into the component's buffer. The activity is composed of two main tasks: i) obtaining the data form the sensor and storing it into the buffer (*ReadData*, *ReadSensorData*, *InsertData*, *SetData*); and ii) updating a counter to keep track of the amount of data stored in the buffer (*ReadCounter*, *AddOnePack*, *SetCounter*). It should be noted that actions *ReadData*, *InsertData*, and *SetData* operate on the *Data2Send* attribute of the component (see Figure 9), while actions *ReadCounter* and *SetCounter* operate on the *counter* attribute.

Figure 11 includes the specification of a fault to be injected in the model. The diagram specifies a fault of type WBC1 to be injected in the *ObjectFlow* element connected to the comment. As described in Table II, the fault changes the *ObjectFlow* to a *ControlFlow*, using the *source* and *target* elements as new connection ends. The connection between the *result* and *value* pins, of *ObjectFlow* kind, will then be replaced by a *ControlFlow* connection between the *AddOnePack* and the *SetCounter* actions. Such modifications has two potential consequences: i) the object that is emitted on *result* pin will not be able to reach the *value* pin, and ii) the action *SetCounter* is

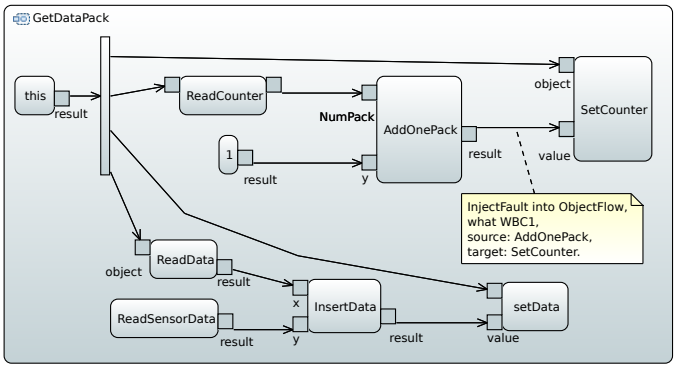

Figure 11: Detail of the *GetDataPack* action, with the specification of the fault to be injected.

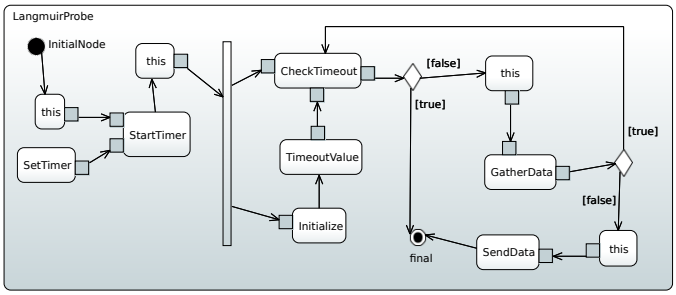

Figure 12: Model of the main behavior of the LP payload, modified to include a timeout for data acquisition.

now forced to be executed right after the *AddOnePack* action. In practice, the second one has no effect in this particular model, since the two actions are already executed in that sequence.

To verify the actual impact of this fault on the overall system behavior, we run the transformation to inject the fault, and then we execute the main behavior with Moka. The injected fault manifest itself as an hang of the *LangmuirProbe* behavior, causing it to enter in an endless loop of calls to the *GatherData* action (see Figure 10). This is caused by the *counter* attribute not being updated anymore (action *SetCounter* does not receive the updated value), which in turn causes the action *GatherData* to never signal that the data acquisition phase has ended.

A possible countermeasure to this fault is to limit the acquisition phase to a certain amount of time, after which a timeout is issued. The modified version of the *LangmuirBehavior* to include a timeout is shown in Figure 12. The timer is initialized (*StartTimer*), then the timeout value is set (*TimeoutValue*), and its expiration is verified at each loop (*CheckTimeout*). In case the activity *CheckTimeout* returns *true*, the activity is terminated. This simple example shows how the proposed approach can be used for uncovering potential flaws in a system model conforming to fUML.

IX. Related Work

The authors of [24] defined a taxonomy of software faults by an accurate analysis of real software faults and categorization. The same taxonomy has been later used by [9] and many other works. We based our work on this taxonomy, in particular for the mapping of the identified fUML faults (Table II).

One of the first works on model-level fault injection was the MODIFI framework [8], which focused on behavioural models specified in Simulink. Conversely, our work focuses on UML models, and specifically on the fUML subset. The InRob approach [31] proposed the extension of state machine models with faulty transitions, as a support to integration and robustness testing. The work in [32] defined mutation operators for Class Diagrams, with the objective to support mutation testing. Other work have addressed fault injection at model level in different ways; however, very few works addressed fault injection on executable fUML models.

By themselves, executable models are not a widely explored topic yet. One of the most general works on model execution in the MDE panorama is the work on xMOF [33]. The authors integrate fUML with MOF, creating a new metamodeling language with support for model execution, called xMOF (eXecutable MOF). xMOF permits specifying both the abstract syntax as well as the execution semantics of any DSL based on MOF. This is the theoretical work behind the Moliz tool mentioned above [20].

Other work has focused on different aspects of executing fUML models. The authors of [34] analyzed version 1.0 of the fUML specification, recently released at that time, highlighting its limits in terms of controllability and observability of execution. The authors then proposed an extension to enable debugging of fUML models. The authors of [35] proposed a modeling and co-simulation environment for CPS, enabling the integrating fUML models in simulation environment based on the Functional Mock-up Interface (FMI). The work in [36] provides an execution solution for UML profiles, formalizing their execution semantics based on fUML. The GEMOC Initiative maintains GEMOC Studio [37], an Eclipse-based framework for the development of heterogeneous executable modeling languages that integrates recent efforts in the area of executable models.

A few works have started addressing testing and verification of non-functional properties on fUML models. The work in [38] proposes an approach to extract runtime information from fUML models, and exploits this information to perform timing analysis directly on the model. The work in [39] and [3] proposes an approach for the semi-automated application of Software FMEA (Failure Modes and Effects Analysis), combining fUML behaviors and Composite Structure Diagrams. That work also uses model-level fault injection, as detailed in [3]. However, fault injection is performed on components interfaces only, i.e., not on the fUML behaviors. In this paper we addressed the injection of faults in fUML models, and in particular in Activity Diagrams.

X. Conclusion

In this work we have proposed an approach for specifying and injecting faults in models conforming to the Foundational UML subset, which allows model execution. This is the first

step towards applying fault injection in fUML models. In this paper, we have demonstrated the applicability of the approach with a simple example in the space domain. As future work, we plan to perform an accurate validation of the proposed approach, both in terms of correctness of the injection process, as well as usability for end users. At this stage, performing an accurate experimental validation appears to be particularly challenging, because the still limited adoption of fUML in the industry.

ACKNOWLEDGMENT

This work has been developed in the context of the H2020-MSCA-RISE-2018 "ADVANCE" project (grant 823788).

REFERENCES

[1] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: a methodology and some applications," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166–182, 1990.

[2] R. Svenningsson, H. Eriksson, J. Vinter, and M. Törngren, "Model-implemented fault injection for hardware fault simulation," *Model-Driven Engineering, Verification, and Validation, Workshop on*, vol. 0, pp. 31–36, 10 2010.

[3] V. Bonfiglio, L. Montecchi, I. Irrera, F. Rossi, P. Lollini, and A. Bondavalli, "Software Faults Emulation at Model-Level: Towards Automated Software FMEA," in *1st Workshop on Safety and Security of Intelligent Vehicles (SSIV 2015)*, Rio de Janeiro, Brazil, 2015, pp. 133–140.

[4] R. Saracco, "Digital Twins: Bridging Physical Space and Cyberspace," *Computer*, vol. 52, no. 12, pp. 58–64, December 2019.

[5] D. C. Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[6] Object Management Group, "Semantics of a Foundational Subset for Executable UML Models (fUML)," formal/2011-02-01. v1.0, January 2011.

[7] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. Leber, "Comparison of physical and software-implemented fault injection techniques," *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1115–1133, 2003.

[8] R. Svenningsson, J. Vinter, H. Eriksson, and M. Törngren, "MODIFI: A MODel-implemented fault injection tool," in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 210–222.

[9] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On Fault Representativeness of Software Fault Injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2013.

[10] V. Slåtten, P. Herrmann, and F. A. Kraemer, "Chapter 4 - Model-Driven Engineering of Reliable Fault-Tolerant Systems—A State-of-the-Art Survey," in *Advances in Computers*, A. Memon, Ed. Elsevier, 2013, vol. 91, pp. 119–205.

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

[12] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[13] J. Bézivin, "On the unification power of models," *Software and Systems Modeling*, no. 4, pp. 171–188, 2005.

[14] Object Management Group, "OMG Unified Modeling Language (OMG UML), Version 2.5.1," formal/2017-12-05, December 2017.

[15] ——, "OMG Meta Object Facility (MOF) Core Specification," formal/2019-10-01. Version 2.5.1, October 2019.

[16] ——, "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification," formal/2016-06-03. Version 1.3, June 2016.

[17] B. Rumpe, *Modeling with UML*. Springer International Publishing, 2016.

[18] C. Starrett, "xtUML: Current and Next State of a Modeling Dialect," in *Proc. of the 2nd International Workshop on Executable Modeling (EXE 2016)*, vol. 1760, Saint-Malo, France, October, 3rd 2016, pp. 33–37.

[19] Model Driven Solutions, "Foundational UML (fUML) Reference Implementation," v1.4.0 (conforms to fUML 1.4), January 2019, http://modeldriven.github.io/fUML-Reference-Implementation/.

[20] T. Mayerhofer and P. Langer, "Moliz: a model execution framework for UML models," in *Proceedings of the 2nd International Master Class on Model-Driven Engineering Modeling Wizards (MW'12)*. ACM Press, 2012.

[21] Object Management Group, "Precise Semantics of UML Composite Structure (PSCS)," formal/2019-02-01. Version 1.2, June 2019.

[22] Z. Micskei, R.-A. Konnerth, B. Horváth, O. Semeráth, A. Vörös, and D. Varró, "On Open Source Tools for Behavioral Modeling and Analysis with fUML and Alf," in *1st Workshop on Open Source Software for Model Driven Engineering*, Valencia, 2014.

[23] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[24] J. A. Duraes and H. S. Madeira, "Emulation of Software Faults: A Field Data Study and a Practical Approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 849–867, November 2006.

[25] H. Bruneliere, J. Cabot, F. Jouault, and F. 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)*, 2010, pp. 173–174.

[26] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi, "Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework," *Software & Systems Modeling*, vol. 15, no. 3, pp. 609–629, May 2016.

[27] L. Bettini, *Implementing Domain Specific Languages with Xtext and Xtend*, 2nd ed. Packt Publishing, 2016.

[28] C. L. G. Batista, E. Martins, and M. de Fatima Mattiello-Francisco, "On the use of a failure emulator mechanism at nanosatellite subsystems integration tests," in *2018 IEEE 19th Latin-American Test Symposium (LATS)*, March 2018.

[29] N. J. Schuch *et al.*, "The Present & Future of the Brazilian INPE-UFSM NANOSATC-BR, CubeSats Development Program," *Annales Geophysicae Discussions*, vol. 2019, pp. 1–16, 2019.

[30] D. P. de Almeida, "Modelagem da Interoperabilidade entre Computador de Bordo e Cargas Úteis do NanosatC-Br2 com apoio da Ferramenta UPPAAL e Análise & Projeto da Daughterboard do Computador de Bordo," Master's thesis, Universidade de São Paulo, 2016.

[31] F. Mattiello-Francisco, E. Martins, A. R. Cavalli, and E. T. Yano, "InRob: An approach for testing interoperability and robustness of real-time embedded software," *Journal of Systems and Software*, vol. 85, no. 1, pp. 3–15, January 2012.

[32] M. F. Granda, N. Condori-Fernández, T. E. J. Vos, and O. Pastor, "Mutation Operators for UML Class Diagrams," in *Advanced Information Systems Engineering (CAiSE)*. Springer, Cham, 2016, pp. 325–341.

[33] T. Mayerhofer, P. Langer, and M. Wimmer, "xmof: Executable dsmls based on fuml," vol. 8225, 10 2013.

[34] Y. Laurent, R. Bendraou, and M.-P. Gervais, "Executing and debugging UML models: an fUML extension," in *SAC'13 - The 28th Annual ACM Symposium on Applied Computing*. Coimbra, Portugal: ACM, Mar. 2013, pp. 1095–1102.

[35] S. Guermazi, S. Dhouib, A. Cuccuru, C. Letavernier, and S. Gérard, "Integration of UML models in FMI-Based co-simulation," in *2016 Symposium on Theory of Modeling and Simulation (TMS-DEVS)*, 2016, pp. 1–8.

[36] J. Tatibouët, A. Cuccuru, S. Gérard, and F. Terrier, "Formalizing execution semantics of uml profiles with fuml models," in *Model-Driven Engineering Languages and Systems*, J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, Eds. Cham: Springer International Publishing, 2014, pp. 133–148.

[37] B. Combemale, O. Barais, and A. Wortmann, "Language Engineering with the GEMOC Studio," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, April 2017.

[38] L. Berardinelli, P. Langer, and T. Mayerhofer, "Combining fUML and Profiles for Non-Functional Analysis Based on Model Execution Traces," in *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures (QoSA'13)*. New York, NY, USA: ACM, 2013, p. 79–88.

[39] V. Bonfiglio, L. Montecchi, F. Rossi, P. Lollini, A. Pataricza, and A. Bondavalli, "Executable Models to Support Automated Software FMEA," in *16th IEEE International Symposium on High Assurance Systems Engineering (HASE 2015)*, Daytona Beach Shores, FL, USA, 2015, pp. 189–196.