Software Faults Emulation at Model-Level: Towards Automated Software FMEA

Valentina Bonfiglio[1], Leonardo Montecchi[1,2], Ivano Irrera[3], Francesco Rossi[4], Paolo Lollini[1], Andrea Bondavalli[1,2]

[1] University of Florence – Firenze, Italy – {vbonfiglio, lmontecchi, lollini, bondavalli}@unifi.it
[2] Consorzio Interuniversitario Nazionale per l'Informatica (CINI), University of Florence – Firenze, Italy
[3] University of Coimbra – Coimbra, Portugal – ivano@dei.uc.pt
[4] ResilTech s.r.l. – Pontedera, Italy – francesco.rossi@resiltech.com

***Abstract*— Safety is a fundamental property for a wide class of systems, which can be assessed through safety analysis. Recent standards, as the ISO26262 for the automotive domain, recommend safety analysis processes to be performed at system, hardware, and software levels. While Failure Modes and Effects Analysis (FMEA) is a well-known technique for safety assessment at system level, its application at software level is still an open problem, especially concerning its integration into certification processes. Fault injection has been envisioned as a viable approach for performing Software-FMEA (SW-FMEA), but it typically requires an advanced development stage where code is available. The approach we propose in this paper, aims to perform software fault injection at model-level, namely on fUML-ALF models obtained from a component-based UML description through transformations proposed in a previous work. Model-level fault injection allows SW-FMEA to assess the effectiveness of safety mechanisms from the early stages of system design. The work in this paper focuses on how the software fault injection is implemented, and on the study of fault propagation through appropriate points of observation to highlight possible violations of requirements, with the identification critical paths.**

Keywords—software safety analysis; executable model; ALF; fUML; component-based; model-implemented fault injection.

I. Introduction

Software is becoming increasingly important in the design of safety-critical systems, thus impacting on safety requirements, which must be assessed at both hardware and software level. For this reason, recent safety standards are putting more and more emphasis on safety analysis at software-level. In the automotive domain, the recent ISO26262 standard [3] for the functional safety of road vehicles foresees safety analysis to be performed at different levels: system, hardware, and *software*. However, besides software safety analysis being mandatory at various levels (e.g., architectural level), such standards do not specify how it should be performed, thus resulting in a gap to be addressed for its industrial adoption.

In our previous work [1] we have proposed a high-level view of the activities that are needed to perform a rigorous safety analysis in accordance with Part 6 related to Product development at the SW level and Part 9 related to Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses of ISO26262 standard [3]. The proposed activities include the **definition or refinement of a SW model**, the **definition of a fault model**, and the **application of the SW FMEA technique**, among other support activities. A high-level view of the activities workflow is illustrated in Fig. 1. Our subsequent work focused on defining an approach to perform automated SW-FMEA at model-level. In fact, Failure Modes and Effects Analysis (FMEA) is an important step in any safety analysis, and its application at hardware and system levels has been extensively addressed in the literature. Conversely, an equivalent analysis at *software level* has not such an established background. In particular, in [2] we introduced an approach for SW-FMEA based on executable fUML models, obtained by model-transformation starting from a component-based system architecture described in a syntactically richer UML model.

In this paper we take the successive step towards automatic SW-FMEA, proposing the use of the fault injection technique at model-level to emulate the faulty behavior that the system can present. Possible faulty behavior is detected by executing the system model and comparing the system's nominal behavior with its behavior when faults are injected. We believe that model-execution and model-level fault injection can facilitate the highlight of faulty behavior, which can be used for: i) studying the propagation of faults for identifying high-risk components, ii) identifying and implement appropriate fault mitigation mechanisms, and iii) validating such safety mechanisms.

Furthermore, besides the fact that several fault injection techniques can be found in literature [8][9][17][18][19][28], there are few works that address the problem of injecting at

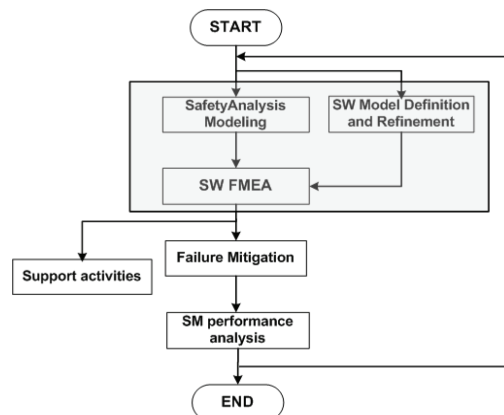

Fig. 1. Safety analysis of software architectures according to ISO26262 [1].

model-level; these work do not specify the fault model associated to the injection and its representativeness (i.e., if the injected faults are likely to occur in a real system). In this paper, we propose to inject the expected effect of a fault at the interface of components, thus emulating failures at the component interface. Our approach permits to i) address the problem of specifying a fault model at the architectural level, where software still does not exist, and ii) study the propagation of faults as the effects of failures propagating from a component to another.

The paper is organized as follows. Section II discusses the background on semi-automatized SW-FMEA and the related work about fault injection addressing FMEA analysis. Section III presents the semi-automatized SW-FMEA approach defined in [2], and how the executable system model can be obtained from its use. Section IV describes the proposed fault injection approach, including the identified fault model. The implemented framework is then presented in Section V. Finally, concluding remarks are reported in Section VI.

II. Background and Related Work

In this section we give some background on model execution and on fault injection, which stand at the basis of our approach for SW-FMEA.

A. *Execution of UML models*

The execution of UML models is a prerequisite to our workflow. The topic has attracted much attention in recent past. For example, UML/P [37] is an implementation-oriented variant of UML allowing code generation [38]. A standard semantics for the execution of UML models, called Foundational UML (fUML), has been published for the first time only in February 2011 by the Object Management Group [4]. Since then, work on the execution and simulation of fUML models has started to emerge. However, execution of UML models for safety analysis is still a relatively new topic. To the best of our knowledge, the most complete proposal for fUML execution is Moliz [39]. Moliz supports the test and validation of UML models; however, its focus is on functional testing of (f)UML models, i.e., behavior in presence of faults is not specifically addressed.

The Action Language for Foundational UML (or "ALF") [5] has been defined by OMG as the surface notation for specifying executable fUML behaviors within a broader model primarily represented by the usual UML notation. In our work we use the ALF representation of fUML models; for this reason the terms ALF and fUML will be used as synonyms in the rest of this paper.

It should be noted that several works have adopted model-transformation to perform different tasks related to dependability and safety analysis [40]. For example, in [42] faults and their effects are modeled at UML level in order to analyze error propagation and testability. In [41] UML annotations are exploited to perform quantitative dependability analysis from the early phases of system design. Differently from most existing work in this domain, in our workflow (see also [2]), transformations are applied to obtain an executable UML model, rather than some dependability-specific analysis model (e.g., fault trees).

B. *Fault injection*

Faults are defined as the hypothesized cause of an error (an unexpected internal state of a system) that can lead to a system failure (e.g., crash, performance degradation, or any interruption of the service provided by the system) [7]. In the context of computer systems, faults can be divided in *hardware faults*, occurring in hardware components (e.g., a bit-flip in a system register due to excessive radiation), and *software faults*, defects in a piece of software that exist due to some issue during the development phase.

A well-known approach for analyzing systems in the presence of faults is *fault injection*, which consists of deliberately inserting faults into a system in a way that emulates faults present in the system [8]. Fault injection techniques have been used in several scenarios: validation of fault tolerance mechanisms implemented on a system, dependability validation [9], [10], estimation of fault tolerance parameters (e.g., fault coverage and error latency) [8], dependability benchmarking [11], and failure prediction [12][13]. Besides hardware faults, during the last decades it has been demonstrated that *software faults* became the major cause of computer systems' failures ([14], [15]), also due to the increasing complexity of systems' software. The injection of software faults was first addressed in [16], but many other works have been developed later (e.g., [8], [17], [18]). The aim of *software fault injection* is to emulate residual faults, i.e., faults that escaped the testing phase at different system's development levels. In fact, several works focusing on the importance of the impact of software faults onto the safety properties of a critical system came up in the last years [19].

In the context of Failure Modes and Effects Analysis (FMEA), several works adopted fault injection techniques to emulate systems failures and execute the FMEA in an automatic manner (e.g., [20], [21]), both injecting in the system under test and in a high-level model of the system. Other works also used fault injection and FMEA in the development phase to improve the architecture of a system in terms of fault tolerance, thus in a perspective of co-designing [22]. Nonetheless, the existing works aimed at injecting mainly hardware faults for modeling system's failure modes, not considering software as a source of failures, which is useful for performing FMEA at software level. Among the first works in SW-FMEA, Ammar et al. [23]–[25] proposed a framework for performing SW-FMEA and modeling the impact of software defects at design level on the rest of the components. The system model under test is injected with several faults, i.e., errors the designer could have done during model design. The authors base their framework on the use of UML-RT for modeling a real-time system, and the fault model defined is based on UML-RT (structural and behavioral) elements. In particular, the authors show that their approach is able to help in performing a SW-FMEA for real-time systems as a Pacemaker, whose model is analyzed. Nonetheless, the analysis comprises the presence of design defects only, and no analysis on the validity of the injected defects (e.g., what is the

probability that a certain defect is present in the model) is performed. Snooke et al. in [26] aim to automate SW-FMEA for safety-critical embedded systems and model-driven software developments, similarly to [22], proposing automatic fault propagation model construction from the software, the use of software fault injection in the model, and the identification of system level effects. Nonetheless, the authors do not distinguish faults between software defects, hardware defects or system errors.

In the software safety analysis scenario, there is the problem of emulating software faults that can be present in system models in a realistic way, thus the faults representativeness must be addressed. However, in the works presented above the representativeness of the injected faults was not addressed. Cotroneo and Natella in [19] propose the injection of realistic software faults, on the basis of the most occurring coding faults when developing a system. Nonetheless, the injection of software faults requires the existence of the software system, thus making such approach not usable at architectural and model-level. Moreover, literature is also plenty of error injection techniques, emulating the activation of software faults [27]–[30]; however their applicability at model-level is limited as well.

Conversely, few works on failures injection can be found. The cause of the failures injected, i.e., the actual faults, can be both hardware (soft faults) and software failures, being independent on the development phases. Generic failure models were proposed by Bondavalli and Simoncini [31] (timing and value failures), extended by McDermid and Pumfrey in [32] and used in [33] (adding the "commission" failure, and thus the failure dimension relative to "Service Provisioning"). Such failure models were adopted by Wallace et al. in [6], for assessing fault propagation in a proposed architectural model. Few other failure modes exist in software systems literature (as for instance the Koopman and DeVale C.R.A.S.H. scale for OSs [34]), but their adaptation to this scenario is not interesting for now.

III. Automated SW-FMEA Based on Executable Models

In this section we recall the approach for performing SW-FMEA using executable models and fault injection that was proposed in [2], while the approach for injecting faults at model-level and the proposed framework are presented in the next sections. In such work, we have explored a practical approach to perform SW-FMEA at model-level (i.e., to cover the part enclosed in the box in Fig. 1). In particular, we proposed a modeling approach and a workflow for performing SW-FMEA, based on model execution and foreseeing the use of fault injection.

The workflow firstly creates a component-based description of the software architecture (Fig. 2), which includes behavioral information to be used in the model-execution phase to simulate the runtime behavior of the system. At this stage the software model is a functional, (ideally) fault-free description of the system (or *golden* model). Successively, our hypothesis is to have a specific engine able to interpret such (architectural, functional and behavioral) model, which can be executed, obtaining execution traces that are representative of the behavior of the system under analysis. In particular, the *golden* model will produce a *nominal* behavior. At last, fault injection can be used to define faulty versions of the model, whose execution can evidence faulty behavior, and data can be used for safety analysis.

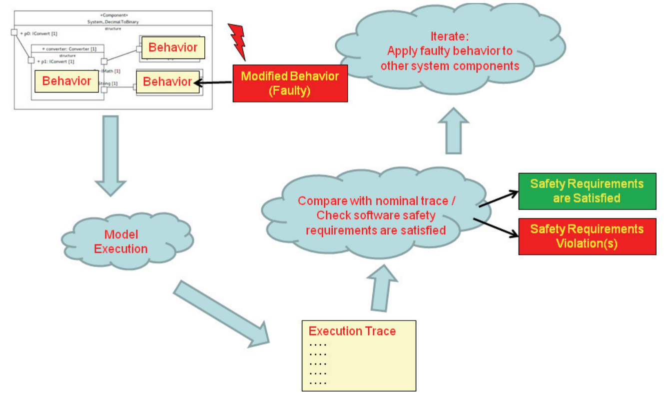

Fig. 2. The detailed workflow to perform the SW FMEA activity.

The first step was to define an approach for system modeling, giving also the possibility to perform model execution. With respect to other model-execution approaches, our focus is on component-based UML models, which supports a general-purpose description applicable to a wide range of application domains. Unfortunately, the Object Management Group (OMG) intentionally omitted a variety of useful constructs out of the standard for UML execution, fUML [4], to restrict it to a small set of *foundational UML* elements with precise semantics. Some of the core elements of component-based descriptions (e.g., components, ports, connectors) are among the excluded elements. In [2] we overcame this limitation by defining a model-transformation capable to generate executable fUML-compliant models from a component-based system architectures described in a syntactically richer UML model.

We hypothesize that the systems can be modeled with respect to its functionalities, which are associated to operations at component level, the way in which communication (through ports) occur between components (push or request/response), and optionally the duration of execution of operations. Finally, the system includes a scheduler for the management of operations execution and timing: selected components' operations are executed cyclically, with well-defined phase and period. The failures, in particular, will be defined based on such model.

Practically, our component-based UML representation uses elements from: i) *UML*, to model components, ports, and composite structures; ii) *fUML* (in its ALF representation), to specify the behavior associated to operations; and iii) *MARTE* [1], to specify the periodicity of execution of operations, and to distinguish ports with different interaction kinds. In such *component-based* architectural model two kinds of interactions may exist between components: *data flow* and *function call* interactions. *Data flow* concerns with data exchange, e.g., a value produced by component A is used as an input by component B; *Function call* identifies the invocation of an

TABLE I. EXAMPLE OF ALF CODE [5]

```
/* Generic component */

  public active class C {
    public i: Integer;
    public j: Integer;
    public t: TestRunner;
    @Create public C(in i: Integer, in t: TestRunner) {
      this.i = i;
      this.t = t;
    }
  }
```

operation, e.g., component B calls a function exposed by component A. Component instances are connected to each other and interact only through their (compatible) *ports*. Two ports are said to be compatible if they are of the same type, and of opposite direction (input/output). Each component may have a set of *operations*. In our approach, we specify the behavior of the operations at design-level (i.e., without coding language- and hardware-related details) using the ALF language. A subset of such operations may be exposed for function call interactions; such operations are *interface operations* of the component. In addition to being called by other components, we assume that operations defined by components can be executed by the environment, i.e., the scheduler, the OS, or in general the execution platform. Accordingly, we allow the model to define the periodic execution of a set of operations.

Such component-based representation is then automatically transformed into an executable model in the ALF language, thus making it executable. As example, we present in Table I an example of ALF code resulting from the transformation. It is worth noting that the code of the whole system is executed through a single entry point, here also referred to as *ALF main*.

IV. THE PROPOSED FAULT INJECTION APPROACH: FAILURE INJECTION IN ALF MODELS

In this section, we present the proposed approach to inject failures in the ALF model of a system. We start from an executable model of the SW architecture that describes the nominal behavior of the system, obtained as in [2]. Our proposal is to modify the model using a library of possible failures, emulating the activation of unknown faults, and then to execute the model to observe error propagation and the impact of the injected failures on the overall SW architecture, possibly resulting in the violation of safety requirements.

A. *Failure injection: operation -level vs component-level*

In defining a fault injection approach for the methodology proposed in [2], the first question faced was, considering the component-based software architecture, where to inject failures for emulating the presence of faulty components. The possible choices identified are:

- the *component-level,* i.e., considering a component as faulty and modeling the possible failure modes it might present at its boundaries;
- the *operation-level* (in the present work described in ALF), in which failures can be injected directly in the ALF specification of operation bodies, thus emulating design faults, e.g., wrong behavior of the component at the design level.

TABLE II. EXAMPLE OF MODIFIED ALF CODE AND TRIGGER USAGE

```
/* Generic component */

  public active class C {
    public i: Integer;
    public j: Integer;
    public t: TestRunner;
    @Create public C(in i: Integer, in t: TestRunner) {
      this.i = i
      if (trigger_001 == true) {
        this.t = t+1;            //incorrect value
      }
      else {
        this.t = t;
      }
    }
  }
```

Since we are most interested in the effects of component failures on the rest of the architecture, we inject failures at component’s boundaries. For this reason, as the injected failures would be always active (i.e., an injected failure would always be executed), we introduce the use of *trigger*, for simulating the activation of a fault and its evolution to a failure, according to the model in [7]. A simple implementation of a trigger can be an *if(condition)* at ALF code-level, where the condition is associated to a given time or a given input to the model under execution.

It should be noted that injection can be performed directly on the executable ALF model generated by the transformation, or as part of the model-transformation algorithm itself. Our current implementation, detailed in the following, focuses on the first option.

B. *Failure injection and failure emulation triggers*

A failure is injected in the ALF code by modifying the code according to a given fault type. However, as already said, for emulating the activation of a fault and its evolution to a failure, the strategy we adopted here is based on the use of *triggers*. A trigger is a variable inserted in the code, wrapping the fault injected; in particular, as a fault is emulated by adding, deleting or modifying existing ALF code, such part of the component’s code is wrapped. An example is presented in Table II.

On the basis of the value of the trigger, a part of code different from the nominal will be executed in order to simulate the insertion of the considered fault. The proposed fault injection approach works offline, injecting/modifying the ALF code of all the components of the system. With such approach, all the faults can be injected at once, while they can be activated one at a time.

C. *Failure model*

The proposed failure model is based on the models proposed in [6][31][32], in which failures are considered as perturbations in the service that the system (or also a *component*) offers (as defined also in [7]), e.g., a system can provide an incorrect value (service value failure), output a value too late (service time failure), or output anything (service provisioning failure). A failure can be modeled according to three dimensions and relative cases listed in the following:

- **Service provision**: omission, commission;
- **Service timing**: early, late;
- **Service value**: coarse incorrect, subtle incorrect.

In particular, the adopted model reflects the classification of failures by Bondavalli and Simoncini [31], later extended by McDermid and Pumfrey [32], who organized the failure modes in a slightly different way, and added the *commission* failure mode, as in that model there was no mode representing a spurious output from a component. However, despite this evolution, we did not adopt the *commission* failure mode in our model. The presented failure model is applied to the ports of the ALF components, which are divided in Data Flow ports (push) and Client/Server ports (request/response) (MARTE profile [1]), and to the timing of the components' operations. In this paper we consider only the failure modes ***coarse incorrect***, ***subtle incorrect***, ***omission***, ***early*** and ***late***, for the sake of simplicity, but expecting extensions in the near future.

The failure model we present in this work is defined on the flow ports and client/server ports, besides the system timing, and is presented in Table III. In particular, the table presents the failure modes associated to each port, and the way in which the injection can be performed in the *ALF* code. Each kind of failure is described by i) the type of *model element* corresponding to the ALF code to which it can be applied, ii) the *failure mode* that can be injected, iii) how the failure is *emulated* in the ALF code, and iv) the identifier of the *trigger* used for activating the failure.

TABLE III. FAILURE MODEL, EMULATION PATTERNS AND TRIGGERS

Type of port / Property	**Fault/ Failure Type**	**Emulation Pattern**	**Trigger Type ID**	**Trigger ID (specific to the single injected failure *x*)**
FlowPort – Output	Incorrect value – in range	the output of a component is wrapped, and the value of the port is changed to a known value when the trigger is activated	1	*1_x, x={1, 2…}*
FlowPort – Output	Incorrect value – out of range	same as above	2	*2_x, x={1, 2…}*
FlowPort – Output	Omission value	same as above, removing the output value	3	…
Timing of internal operation	Incorrect phase – Early	wrapper + trigger + operation phase altered (scheduler)	4	…
Timing of internal operation	Incorrect phase – Late	wrapper + trigger + operation phase altered (scheduler)	5	…
Timing of internal operation	Incorrect period – Early	wrapper + trigger + operation period altered (scheduler)	6	…
Timing of internal operation	Incorrect period – Late	wrapper + trigger + operation period altered (scheduler)	7	…
Scheduler	Incorrect variable value	wrapper + trigger + scheduler operations order altered	8	…
ClientServer Port	Incorrect Call	the call to another component is wrapped and its code changed to another operation, or the same operation but other parameters	9	…
ClientServer Port	Omission Call	same as above, removing the operation call	10	*10_x, x={1, 2…}*

For what concerns the **FlowPorts**, they can be input or output ports. In this work we do not consider injecting faults at the input of a component, as the output flowports are input flowports respect to another component connected to it. This consideration is important because a value that is correct at the output of a component can be altered at the input of a component by an erroneous conversion or marshaling, for instance. FlowPorts may be injected with:

- **incorrect value – in range**, the value at the output of the flowport is incorrect, but it is within the range of valid values;
- **incorrect value – out of range,** the value presented is out of the range of valid values;
- **omission value,** the flowport does not provide any value, when expected to do so.

About **ClientServerPorts**, two kinds of faults are considered. The *Incorrect Call* fault indicates either i) a call to an incorrect operation, or ii) a call to the correct operation, but with incorrect parameters. The *Omission Call* fault instead means that the call to an operation is not executed when expected.

Some of the faults in Table III are also related to the timing aspects. In particular:

- **Incorrect period – Early**, the execution of the operation is repeated with a period smaller than the nominal one, i.e. more frequently.
- **Incorrect period – Late**, the execution of the operation is repeated with a period larger than the nominal one, i.e. less frequently.
- **Incorrect phase – Early**, the execution of the operation starts earlier with respect to the established instant of time.
- **Incorrect phase – Late**, the execution of the operation starts late respect to the established instant of time.
- **Scheduler –** The order of the execution of the operation, fixed by the scheduler, is altered.

D. Faultload and workload

After defining the failure model, it is necessary to define a *faultload* containing the failures to inject in the system's model, that is, it is necessary to specify *what*, *when*, and *where* a failure must be injected. After this a workload represents a typical execution profile for the considered application area must be defined. The selection of valid workload and faultload is of utmost importance for a valid failure modes analysis. However, we do not specify any particular workload and faultload here, as it goes beyond the scope of this work.

V. THE SW-FMEA FRAMEWORK

The approach for performing an automatic SW-FMEA based on the injection of failures as defined in the previous section is summarized in Fig. 3 and organized in a framework. The steps the SW-SMEA framework proposed here takes for assessing

safety hazards of a system's fUML/ALF model can be summarized as follows:

1. **modeling**: the system architecture is described according to the component-based model proposed in [2], where each component provides a set of operations (each with its ALF implementation). The failure model, i.e., the failures to inject, and the observation points can also be defined at this level;
2. **model transformation**: the model transformation proposed in [2] is executed, thus obtaining a global executable ALF model; such model uses only constructs in the fUML specification it is therefore executable. Such model contains a "main" Activity derived from the periods and phases of execution of operations executed by the execution platform, and is the entry point of the simulation;
3. **failure injection**: faults from a pre-defined library are injected into the model; the library may be customized based e.g., on i) the kind of system under analysis, or ii) preferences or common practices of industrial actors. Failures are injected at the interface of components, thus emulating software faults. Each failure is associated to a trigger, which allows the emulation a generic software fault in correspondence to given events (e.g., the component being executed) or policies (e.g., emulating fault bursts). It should be noted that, in principle, fault injection can be performed either separately (as in this paper), or during the model-transformation process (step 2);
4. **observation points**: the specified observation points are added to the ALF code, logging information as the execution of the components, the call of a given function, and so on. Such logs will constitute the model's execution trace;
5. **model execution**: the model is executed, and an execution trace is obtained. This involves executing both the nominal ("gold") model, as well as the faulty mutations obtained through fault injection.
6. **results analysis**: the execution traces(s) are then analyzed in order to evaluate the effectiveness of safety mechanisms, or detect requirements violations.

While in [2] we focused on steps 1 and 2 from the above list, the present paper focuses on steps 3 to 6, that is, we specify the fault injection component to be integrated in the workflow presented above. In particular, the problem of injecting (or emulating the existence of) software faults at development-time and model-level is that the system is not yet implemented. Therefore, we propose to perform the injection of failures at model-level, in a way to emulate the effect of the activation of a software fault at the interface of the faulty system component. From the perspective of the involved component we are thus injecting *failures* [7] on the interfaces it provides to the other components; however, from the perspective of the overall system architecture they are *faults*. For this reason, in this work the term *fault injection* coincides with the proposed *failure injection*, where not explicitly specified.

TABLE IV. TRIGGERS TABLE

Trigger ID	Trigger Type ID	Start	Duration	Value	Comment
FP_A_001	1	100	1	1	an instant of time
FP_A_002	1	100	200	1	an interval
FP_A_003	1	1	500 (*)	1	always

The proposed framework is composed of three main modules: i) an **injector**, that injects the faults offline, ii) a model simulation **launcher**, which also activates one or more injected faults in each execution, and iii) a **checker**, that searches for safety requirement violation or other conditions. A comprehensive visual description of the framework components interactions is presented in Fig. 3.

A. Design-time definitions

The first steps are the definition of the failure model, and the translation of the safety requirements in rules. In particular, the aim of the rules is to check the safety requirements validity. Thus, a designer should translate requirements in machine-readable information, so that it can be processer by a *Checker*, using information coming from model execution. Checking can be performed offline, applying the rule to one or more model outputs, or online, collecting the model outputs on-the-fly and applying the rules to a selected subset of observation points.

B. The fault injection, triggers and faultload

As discussed in the previous section, the faults/failures are inserted directly in the system's ALF code model, and a trigger is associated to each fault. The proposed fault injector works offline, injecting/modifying the ALF code of all the components of the system, and putting a trigger around each fault injected. In this case, a single or multiple faults can be activated based on their trigger. The triggers are saved in a table, called *triggers set* (an example is in Table IV).

A selection of the triggers will correspond to a *selection* of the faults/failures to inject, and thus to a *faultload*. Each failure activation/trigger has also associated a *starting time*, a *duration* of the failure (if applicable), and a *value*, in the case the failure is related to an incorrect output, for instance.

The structure of the triggers table is presented in Table IV, while an example of faultload is presented in Table V. In particular, Table V presents a faultload with three possible

TABLE V. FAULTLOAD CONTAINING THREE FAILURES TO INJECT

Trigger ID	Component	Port	Fault/Failure Type
FP_A_001	Component_1	FlowPort_Out_A	Incorrect value – out of range
FP_A_002	Component_1	FlowPort_Out_A	Incorrect value – out of range
FP_A_003	Component_1	FlowPort_Out_A	Incorrect value – out of range

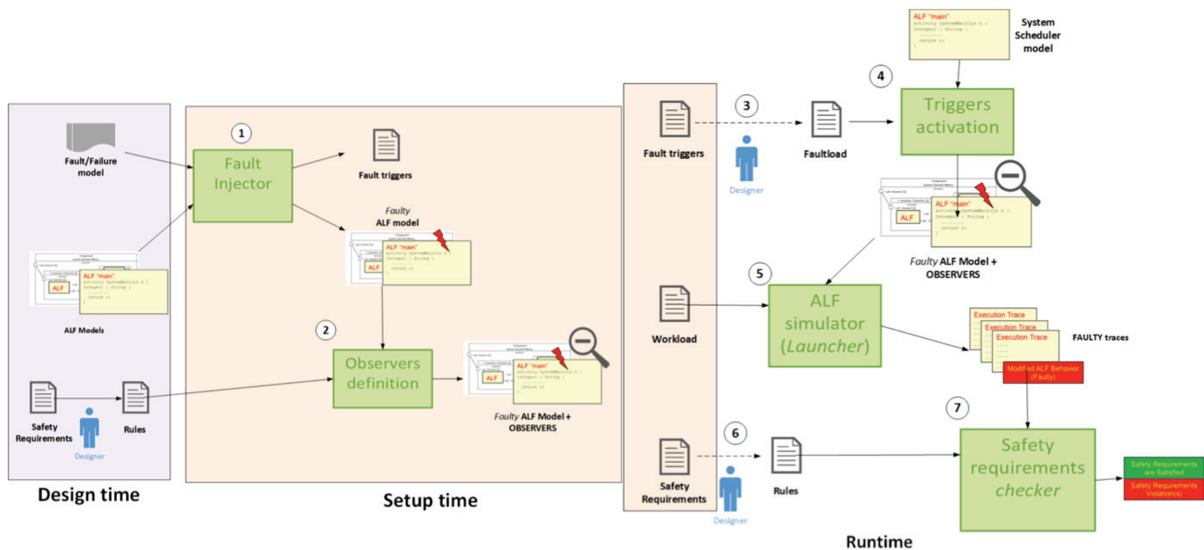

Fig. 3. Detailed view of our approach for automated SW-FMEA, based on fUML/ALF model execution and failure injection.

failures, whose triggers differ from each other by their application time (start) and duration (see Table IV).

To support the proposed approach we implemented an automatic tool for injecting faults in the ALF model of a system, based on the defined faultload. In particular, the injector is implemented in Python and takes ALF code files in input, from which it searches injection points and injects one or more injection types from Table III. Besides this, the injector builds a table for keep trace of the triggers associated to each fault, each of them identifying a single fault/failure.

C. *Observation points*

In order to identify the critical path and study the fault propagation, it is necessary to introduce some *Observation Points*, i.e., points in the code outputting information about variables (for example a output FlowPort of a component) that need to be monitored. The choice of such observation points should be done on the basis of the considered safety requirements. The aim is to make it possible to obtain an execution trace that contains the values of the chosen observation points, as result of the execution of the generated code. After defining the observation points, the *Rules* may be defined based on the information collected by them.

D. *Launcher*

The launcher is responsible for the start of the ALF simulation, using the ALF reference implementation [5]. An execution of the model can be a *golden* execution (i.e., without injected faults), generating a *reference trace*, or a *faulty* execution, if some of the injected faults are activated through their triggers. The activation is done on the basis on the faultload table and with the chosen workload (e.g., a given functionality the system must perform).

E. *Checker*

Finally, the checker is a module that takes in input the *rules* that permit to verify if there are some safety requirement violations. The checker also can receive as input one or more model execution output traces.

VI. Concluding Remarks

The importance of safety analysis of software architectures is growing. In particular, the recent ISO26262 standard comprises several requirements on the safety analysis of software. Defining a precise workflow for the assessment of software architectures is therefore of great industrial relevance. Model-execution and fault injection is a promising approach to apply SW FMEA in the early phases of software design. Based on previous work, we have detailed our process of fault injection. We first recalled how the executable model is obtained and its properties; then, we defined a fault library to take as reference during the injection process. Finally, we detailed on the fault injection process and the current implementation of the framework. Next steps are aiming at extending and consolidating this work in several directions, the main one being the integration of the SW-FMEA approach into the CHESS-CONCERTO multi-purpose framework for the design and evaluation of complex systems [35]. Another improvement would consist in embedding the fault injection process directly in the model-transformation from the UML+ALF to the ALF only model. In this way we could take advantage of specialized model manipulation tools (e.g., EMF-IncQuery [36]). Finally, one aspect that deserves further research is the handling of timing concerns, although the purpose here is not to replicate the features of the multitude of analysis techniques existing in the real-time domain.

Acknowledgment

This work has been partially supported by the CECRIS project, FP7–Marie Curie (IAPP) number 324334, by the ARTEMIS-JU CONCERTO project (n.333053), by the TENACE PRIN project (n.20103P34XC) funded by the Italian Ministry of Education, University and Research, by the DEVASSES project, funded by European Union's Seventh

Framework Programme under grant agreement PIRSES-GA-2013-612569, and by the MS-VIVA project, funded by the Tuscany Region within the framework POR CReO FESR.

REFERENCES

[1] V. Bonfiglio, et al. "On the Need of a Methodological Approach for the Assessment of Software Architectures within ISO26262," SAFECOMP 2013 - Workshop CARS (2013).

[2] V. Bonfiglio, et al. "Executable Models to Support Automated Software FMEA," HASE (2015).

[3] ISO 26262 "Road vehicles -- Functional safety" (2011).

[4] fUML Reference Implementation, Accessed at 14/03/2014 http://portal.modeldriven.org/project/foundationalUML.

[5] ALF Reference Implementation, http://modeldriven.org/alf/, Accessed 14/03/2014.

[6] M. Wallace, "Modular architectural representation and analysis of fault propagation and transformation," Electr. Notes Theor. Comput. Sci., 141(3):53–71 (2005).

[7] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[8] J. Arlat, Y. Crouzet, and J. C. Laprie, "Fault injection for dependability validation of fault-tolerant computing systems," in *19th International Symposium on Fault-Tolerant Computing*, 1989, pp. 348–355.

[9] 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," *Softw. Eng. IEEE Trans. On*, vol. 16, no. 2, pp. 166–182, 1990.

[10] J. Duraes and H. Madeira, "Multidimensional characterization of the impact of faulty drivers on the operating systems behavior," *IEICE Trans. Inf. Syst.*, vol. 86, no. 12, pp. 2563–2570, 2003.

[11] J. Duraes, M. Vieira, and H. Madeira, "Dependability benchmarking of web-servers," *Comput. Saf. Reliab. Secur.*, pp. 297–310, 2004.

[12] I. Irrera and M. Vieira, "A Practical Approach for Generating Failure Data for Assessing and Comparing Failure Prediction Algorithms," in *PRDC'14 proceedings*, Singapore, 2014.

[13] M. Vieira, H. Madeira, I. Irrera, and M. Malek, "Fault injection for failure prediction methods validation", in *Proc. of Workshop on Hot Topics in System Dependability* at DSN 2009, Estoril, Lisbon, Portugal.

[14] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer, "Failure data analysis of a LAN of Windows NT based computers," in *18th Symposium on Reliable Distributed Systems*, 1999, pp. 178–187.

[15] I. Lee and R. K. Iyer, "Software dependability in the Tandem GUARDIAN system," *IEEE Trans. Softw. Eng.*, vol. 21, no. 5, pp. 455–467, 1995.

[16] J. Christmansson and R. Chillarege, "Generation of an error set that emulates software faults based on field data," in *Annual Symposium on Fault Tolerant Computing*, 1996, pp. 304–313.

[17] M. C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[18] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Trans. Softw. Eng.*, pp. 849–867, 2006.

[19] D. Cotroneo and R. Natella, "Fault Injection for Software Certification," *IEEE Secur. Priv.*, vol. 11, no. 4, pp. 38–45, Jul. 2013.

[20] L. Grunske, K. Winter, N. Yatapanage, S. Zafar, and P. A. Lindsay, "Experience with fault injection experiments for FMEA," *Softw. Pract. Exp.*, vol. 41, no. 11, pp. 1233–1258, Oct. 2011.

[21] A. Benso, A. Bosio, S. Di Carlo, and R. Mariani, "A Functional Verification based Fault Injection Environment," in *22nd IEEE International Symposium on Defect and Fault tolerance in VLSI Systems, 2007. DFT '07*, 2007, pp. 114–122.

[22] J. Perez, M. Azkarate-askasua, and A. Perez, "Codesign and Simulated Fault Injection of Safety-Critical Embedded Systems Using SystemC," in *European Dependable Computing Conference (EDCC),* 2010, pp. 221–229.

[23] H. H. Ammar, S. M. Yacoub, and A. Ibrahim, "A fault model for fault injection analysis of dynamic UML specifications," in *12th International Symposium on Software Reliability Engineering, 2001. ISSRE 2001. Proceedings*, 2001, pp. 74–83.

[24] S. M. Yacoub and H. H. Ammar, "A methodology for architecture-level reliability risk analysis," *IEEE Trans. Softw. Eng.*, vol. 28, no. 6, pp. 529–547, Jun. 2002.

[25] D. E. M. Nassar, W. Abdelmoez, M. Shereshevsky, H. H. Ammar, A. Mili, B. Yu, and S. Bogazzi, "Error propagation analysis of software architecture specifications," in *Proc. of the International Conference on Computer and Communication Engineering, ICCCE*, 2006.

[26] N. Snooke and C. Price, "Model-driven automated software FMEA," in *Reliability and Maintainability Symposium (RAMS), 2011 Proceedings - Annual*, 2011, pp. 1–6.

[27] R. Chillarege, K. Goswami, and M. Devarakonda, "Experiment illustrating Failure Acceleration and Error Propagation in Fault-Injection". 2002.

[28] M. H. J. Christmansso and M. Rimén, "An experimental comparison of fault and error injection," in *ISSRE*, 1998, p. 369.

[29] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," in *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*, 1989, pp. 340–347.

[30] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri, "An Empirical Study of Injected Versus Actual Interface Errors," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, New York, NY, USA, 2014, pp. 397–408.

[31] A. Bondavalli and L. Simoncini, "Failure classification with respect to detection", *IEEE Comput. Soc. Press*, *1990*.

[32] J.A. McDermid, D. J. Pumfrey, "A development of hazard analysis to aid software design", *in Proceedings of the Ninth Annual Conference on Computer Assurance, 1994. COMPASS '94 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security, June 1994*.

[33] J.A. McDermid, M. Nicholson, D. J. Pumfrey, P. Fenelon, "Experience with the application of HAZOP to computer-based systems"

[34] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, T. Marz, "Comparing operating systems using robustness benchmarks," *Reliable Distributed Systems, 1997. Proceedings., The Sixteenth Symposium on , vol., no., pp.72,79, 22-24 Oct 1997.*

[35] A. Cicchetti et al., "CHESS: a Model-Driven Engineering Tool Environment for Aiding the Development of Complex Industrial Systems" 27th International Conference on Automated Software Engineering (ASE 2012), 2012.

[36] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, D. Varró, "EMF-IncQuery: An integrated development environment for live model queries", Science of Computer Programming, Volume 98, Part 1, 1 February 2015, Pages 80-99.

[37] B. Rumpe, "Modellierung mit UML", Springer Berlin, 2004.

[38] M. Schindler, "Eine Werkzeuginfrastruktur zur agilen Entwicklung mitder UML/P," ser. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[39] T. Mayerhofer, P. Langer. Moliz: A Model Execution Framework for UML Models. In: Proceedings of the 2nd International Master Class on Model-Driven Engineering at MODELS 2012 (2012).

[40] S. Bernardi, J. Merseguer, D.C. Petriu, "Dependability modeling and analysis of software systems specified with UML". ACM Comput. Surv. 45, 1, Article 2 (2012),

[41] A. Bondavalli, et al. "Dependability analysis in the early phases of UML-based system design." Int. J. Comput. Syst. Sci. Engin. 16, 5, 265–275, 2001.

[42] A. Pataricza, et al. "UML-Based design and formal analysis of a safety-critical railway control software module". In Proc. of Symposium Formal Methods for Railway Operation and Control Systems (FORMS03). 125–132, 2003.