Executable Models to Support Automated Software FMEA

Valentina Bonfiglio[1], Leonardo Montecchi[1,2], Francesco Rossi[3], Paolo Lollini[1], András Pataricza[4], 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] ResilTech s.r.l. – Pontedera, Italy – francesco.rossi@resiltech.com
[4] Budapest University of Technology and Economics – Budapest, Hungary – pataric@mit.bme.hu

Abstract—**Safety analysis is increasingly important for a wide class of systems. In the automotive field, the recent ISO26262 standard foresees safety analysis to be performed at system, hardware, and software levels. Failure Modes and Effects Analysis (FMEA) is an important step in any safety analysis process, and its application at hardware and system levels has been extensively addressed in the literature. Conversely, its application to software architectures is still to a large extent an open problem, especially concerning its integration into a general certification process. The approach we propose in this paper aims at performing semi-automated FMEA on component-based software architectures described in UML. The foundations of our approach are model-execution and fault-injection at model-level, which allows us to compare the nominal and faulty system behaviors and thus assess the effectiveness of safety countermeasures. Besides introducing the detailed workflow for SW FMEA, the work in this paper focuses on the process for obtaining an executable model from a component-based software architecture specified in UML.**

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

I. Introduction

Software is increasingly important in the design of safety-critical systems, as more and more safety requirements are assigned to it. Indeed, recent safety standards are putting more and more emphasis on software-level safety analysis. The recent standard ISO26262 for the functional safety of road vehicles foresees safety analysis to be performed at different levels: system, hardware, and *software*.

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* lacks such an established background. Safety analysis of software introduces significant challenges compared to hardware: data on failure modes are typically unavailable as datasheets, and even small changes to the SW architecture can result in significant alterations in the propagation or mitigation of failures.

In a previous work [1] we defined a set of repeatable activities that are needed to perform a rigorous safety analysis in accordance with the ISO26262 requirements. In this paper we further explore the "SW FMEA" activity of such workflow. The ultimate goal is to enable a model-execution and fault-injection based, general purpose, application domain independent, semi-automated SW FMEA on component-based software architectures described in UML. The most important challenges in positioning SW FMEA into the design workflow is that (i) it should be carried out *early enough* to detect possible insufficiencies in the safety mechanisms, thus avoiding large and costly redesign iteration loops; (ii) at the same time, the modeling process has to *advance enough* to assure a proper correlation between the model used for SW FMEA and the actual later implementation.

Our approach suggests the execution of SW FMEA based on the SW architecture model. This level delivers a breakdown of the architecture into components serving as a basis of component integration and implementation. Thus, this is the first step in the design flow in which dependability mechanisms and measures appear and the dependability attributes of the components are already specified. Accordingly, model-execution and fault-injection into architecture models properly facilitate the comparison of nominal and faulty system behaviors in order to: i) verify that the safety mechanisms prevent the violation of safety requirements, ii) identify appropriate mitigation mechanisms, or simply iii) study the propagation of faults.

It is worth to note, that HW fault impact assessment uses a similar approach of starting from high level models since two and half decades [2]. Methods and experiences gained there can be adopted in the future, e.g., checking the faithfulness of fault models at the model and implementation level, by fault injection experiments that share a common statistical root.

Besides SW FMEA this paper details on the process of obtaining an executable model from a SW architecture specification in UML. With respect to other approaches present in the literature, our focus is on component-based UML models, which supports a general purpose approach applicable to a wide range of application domains. Unfortunately, OMG intentionally omitted a variety of useful constructs out of the standard for UML execution, fUML [11], to restrict it to a small set of *foundational* UML elements with precise semantics. The paper presents a way to overcome this limitation by model-transformations capable to generate executable fUML models from a component-based system architectures described in a syntactically rich UML model.

The paper is organized as follows. Chapter II discusses the related work, while Chapter III introduces the context and objectives of the work. Chapter IV describes the derivation of an executable model from a component-based UML architecture, while Chapter V discusses the current prototype. Finally, concluding remarks are reported in Chapter VI.

II. Related Work

SW safety analysis has been addressed in different ways in the literature. A well-known approach is based on failure propagation and transformation annotations: components of the system are analyzed by experts in isolation from the rest of the system, to estimate their behavior in response to potential erroneous stimuli. Their obtained propagation behavior is then integrated into the architectural model, allowing the automatic estimation of the failure behavior of the entire system. Examples of such approach are the FPTC [3] and HIP-HOPS [4]. A limitation of such approaches is the separation of the functionality of components and their failure behavior. Thus models are obtained as result of deep analyses of components' internals, or from hypotheses of domain experts.

Other approaches apply SW safety analysis by means of formal methods, e.g., Event-B [5]. They rely on formal proofs of satisfaction of the designated safety attributes. The main limitation consists in the level of detail that can be achieved, which is typically limited by an exponential increase in complexity with the increase of problem size.

We base our approach on model-execution and model-level fault-injection (FI), which provides the greatest degree of flexibility and can be used in all the stages of SW development. FI has been widely adopted in the literature, and it is also recommended by different safety standards. While initially conceived to introduce faults at a HW level, techniques for fault-injection at SW level have been introduced, collectively known as Software Implemented Fault Injection (SWIFI), e.g., see [15]. Recently, FI at model-level has also appeared, following the overall "model-driven" trend. MODIFI [16] is one of the tools applying model-level FI: it simulates the effect of HW-related faults on Simulink behavioral models. With respect to [16] and to other approaches applying model-execution and FI, we focus on a component-based system design, and use standard UML as a general purpose system design language. Furthermore, our final goal is a comprehensive semi-automated safety assessment process, as described in [1].

The execution of UML models is a prerequisite to our workflow. The topic has attracted much attention in recent past. For example, UML/P [24] is an implementation-oriented variant of UML allowing code generation [25]. Nevertheless, 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 [11]. Since then, work on the execution and simulation of fUML models has started to emerge. However, execution of UML models for safety analysis is a relatively new topic.

The most complete proposal for fUML execution is Moliz [9]. The debugging capabilities, as well as a framework for automated model testing of Moliz support the test and validation of UML models. While Moliz also addresses to some extent non-functional system properties, its focus is on functional testing of (f)UML models.

The Action Language for Foundational UML (or "ALF") [12] has been defined by OMG as the surface notation for specifying executable behaviors within a broader model primarily represented by the usual UML notation. There are three prescribed ways in which ALF execution semantics may be implemented [12]: *Interpretive Execution*, in which the ALF code is directly interpreted and executed; *Compilative Execution*, when the ALF code is translated into a UML model conforming to fUML and then executed; *Translational Execution*, in which the ALF code, is translated into some executable form in a non-UML target language. The latter is for example the approach adopted in [26], where ALF code is translated to C++ code. Under this perspective, the work described in this paper describes a *Compilative Execution* approach.

Several works have adopted model-transformation to perform different tasks related to dependability and safety analysis [7]. For example, in [23] faults and their effects are modeled at UML level to analyze error propagation and testability. In [22] UML annotations are exploited to perform quantitative dependability analysis from the early phases of system design. In this paper, transformations are applied to obtain an executable UML model from a component-based software architecture.

Finally, while this paper is not specific to the ISO26262 standard [14], one of its main motivations stems from the prescriptions of such standard domain. Further details can be found in [1], which fits into a continuously growing area of research activities proposing techniques to facilitate the adoption of the ISO26262 standard in the automotive industry (e.g., see [6] and [8]).

III. Context, Motivation and Overview

This section presents the context and motivation of our approach. The main aim is the definition of a consistent, repeatable, and semi-automated approach to perform safety analysis of SW architectures. Standards prescribe safety analysis of the SW architecture, without however clear guidelines on how it should be performed, thus leaving an important gap for its industrial adoption. This is the case for the ISO26262 standard as well, recently introduced in the automotive domain [14].

A. *Software Safety Analysis within ISO26262*

In a previous work [1] we tried to address this problem by defining a workflow composed of well defined and repeatable activities to guide safety analysis of SW architectures according to the requirements in Parts 6 and 9 of ISO26262.

The workflow (Fig. 1) identifies a set of activities that need to be performed for a rigorous SW safety analysis process. Such activities include the definition or refinement of a SW model, the definition a fault model, and the application of the SW FMEA technique, among other support activities. In this paper we explore a practical approach to cover the part highlighted in gray in Fig. 1.

B. *SW FMEA Workflow Overview*

The core of the workflow of Fig. 1 is the *SW FMEA* activi-

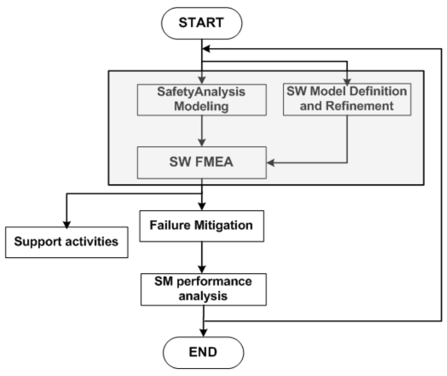

Fig.1. Safety analysis of software architectures according to ISO26262 [1].

ty. An executable model of the SW architecture describes the nominal behavior of the system. The model is altered using a library of possible faults, and then simulated to observe error propagation and failure effects on the overall SW architecture, until a potential violation of safety requirements is observed.

A component-based description of the SW architecture is first created (upper-left part of Fig. 2); such a description should include also behavioral information, to be used in the model-execution phase to simulate the runtime behavior of the system. At this stage the SW model is (ideally) a fault-free description of the system functionality. Such a "golden" model is then executed and/or simulated using a specific engine, obtaining output traces representative of the nominal behavior of the SW architecture under analysis.

Subsequently, the FI phase can start (upper-right part of the figure). 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. Faults are represented in the modified model(s) as deviations from the correct behavior of components, e.g., an incorrect value propagated towards another component of the architecture. Injected faults will then automatically result in faults in the generated executable model, thanks to the transformation algorithm. The model is then executed again with the faulty mutations, and traces of faulty behavior are collected. The traces are then analyzed in order to evaluate the effectiveness of safety mechanisms, or detect requirements violations.

In the rest of this paper we focus on the technological requirements to establish the above mentioned approach. In

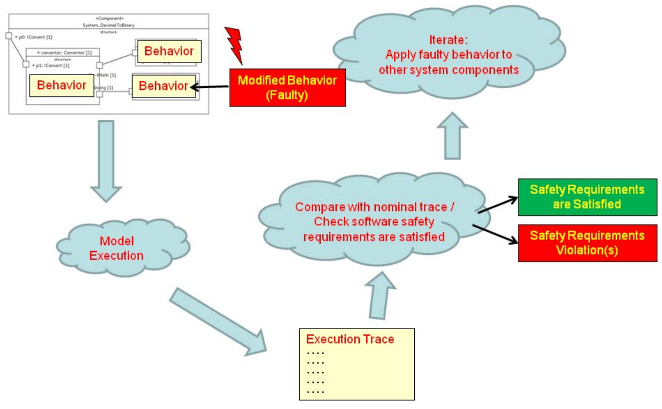
«Component»
System_DecimalToBinary
Behavior
Behavior
Behavior
Modified Behavior (Faulty)
Iterate: Apply faulty behavior to other system components
Model Execution
Compare with nominal trace / Check software safety requirements are satisfied
Safety Requirements are Satisfied
Safety Requirements Violation(s)
Execution Trace

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

other words we will explore i) how the model of the SW architecture should be represented; and ii) how the model of the system architecture can be executed.

C. *Executable UML and Main Gaps*

UML is widely adopted in several domains, and for several very different purposes. Among them, UML is also a common choice as an Architecture Description Language (ADL) [28]. We also base our approach on UML, since it has several practical advantages. First of all its popularity, and the good availability of supporting tools; second, its generality, which allows the same approach to be applied to different domains; last, its powerful extension mechanisms, *profiles*, which allow additional information to be added to already existing models.

UML consists of thirteen diagram types, each adopting a different perspective on the software system under consideration. The diagram types can be categorized into diagrams for modeling the *structure* of a system, and diagrams for modeling its *behavior*. However, in practice only a subset of them is typically used in a given application domain. The UML standard leaves much room for interpretation regarding its semantics: users of UML can develop different variants on their own, which may be inconsistent with each other (e.g,. see [29]). This is obviously a problem for achieving model-execution.

This concern has been addressed by the standardization of fUML, which is a *computationally complete language* [11]. The selected UML subset which constitutes fUML is described as the foundation for eventually defining the execution semantics of higher-level UML modeling concepts. To facilitate the specification of executable models, which can be a very complex task, a textual representation for fUML models is provided by the ALF standard [12], a Java-like representation for fUML elements. Since ALF is a textual representation of fUML, in the rest of the paper we will use the two terms interchangeably.

With respect to the application of our approach, fUML (and thus ALF) has an important limitation, namely the UML elements that are included in the standard [11]. To summarize, fUML consists of *Classes*, for structural modeling, and *Activities*, for behavioral modeling. All the other UML constructs are not included. It should be noted that important modeling constructs like *Components*, *Composite Structures*, *Deployments*, which are the core of component-based modeling approaches, are excluded from fUML. Since we want to stick with a component-based design approach, this is a strong limitation.

Our envisioned solution is to complement a classic component-based UML description of the SW architecture with fUML Activities for specifying behaviors associated with component operations. From this high-level ("surface" [11]) model of the SW architecture, a fUML model should be automatically generated by model-transformation, thus following MDE principles.

In order to successfully apply this approach we need to define: i) constraints that should be imposed in our "surface" UML model; ii) additional properties that may be need to be specified; and iii) the model-transformation algorithm that actually generates the ALF-only model of the system. These aspects are detailed in the next section.

TABLE I. REQUIREMENTS OF THE INPUT ARCHITECTURAL MODEL.

ID	Type	Description
1	FE	The model shall include component names.
2	FE	The model shall include the name, the kind of interaction (i.e., dataflow or function call) and the direction of each port for each component.
3	CO	Ports of component instances must be connected only to ports having the same kind of interaction.
4	FE	The model shall specify a data type for each port representing an interaction of dataflow kind.
5	FE	Each port specifying an interaction of function call kind shall have associated an interface, describing a list of supported operations and their signature.
6	FE	The component model shall include connections between component instances.
7	CO	Ports of component instances describing dataflow interactions can be connected only if their type is the same.
8	CO	Ports of component instances describing function call interactions can be connected only if their interfaces are the same.
9	CO	Ports of component instances can be connected only if they have opposite directions.
10	FE	The model shall allow components to define operations.
11	FE	The model shall allow components to define properties.
12	CO	For each operation defined by a component, an ALF implementation must be provided.
13	FE	The ALF implementation must be able to refer to dataflow interactions of the component, and treat them as variables.
14	FE	The ALF implementation must be able to refer to operations defined on ports specifying function call interactions.
15	CO	The ALF implementation of an operation defined on a component must only refer to elements and attributes belonging to that component.
16	FE	The model may include the duration of each operation defined by components.
17	FE	The model must be able to represent, at system level, the execution period and phase of operations defined by components.
18	CO	The model shall include the execution period of at least one operation.

IV. EXECUTION OF COMPONENT-BASED UML ARCHITECTURES

A. Assumptions and Requirements

The main assumption is that the software architecture is designed using a *component-based* approach: the overall architecture is built from reusable software components. 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. The actual interaction between the two components does not need to be a direct interaction, but it may also arise indirectly, e.g., when component A periodically updates the value of a memory cell, which is periodically read by B. *Function call* identifies the invocation of an operation, e.g., component B calls a function exposed by A. The function may be either a simple "signal" to initiate a certain behavior, or may be an actual function call with parameters and return value.

In such a component-based description, component instances are connected to each other and interact only through their (compatible) *ports*. Two ports are *compatible* if they are of the same type, and of opposite direction (input/output). Furthermore, component instances may be connected together to form more complex components; this can be done recursively at any level of depth. At the same hierarchical level, two component instances A and B may be connected if their ports are compatible; a higher level component may *delegate* one of its ports to a corresponding port of one of its children.

Components of the software architecture define a set of *operations* (i.e., functions), which may have a set of parameters and a return value. A subset of such operations may be exposed for function call interactions; such operations are *interface operations* of the component. Since finally an executable model should be produced, components shall include an ALF specification of their operations (i.e., an ALF "Activity", corresponding to a fUML Activity Diagram). Within such ALF specification it shall be possible to refer to ports of the components as well, either for using them as variables (dataflow interactions), or calling the operations defined on them (function call interactions). Moreover, every ALF specification shall refer only to elements of the involved component, in order to respect the component-based philosophy.

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 assume that the model may specify the periodic execution of a set of operations, each with its own period and phase (i.e., an ordering within the same period). Moreover, each operation can contain a specification of the (worst case) time required for its execution. If such value is not specified, it is assumed that the operation is instantaneous, i.e., its execution time is not of interest for the purposes of the analysis. It should be noted that the purpose for introducing time here is *not* to reason about the schedulability of software tasks, or possible conflicts with respect to the assignment of computational resources. The long-term reason for introducing time is to reason about the consequences of timing failures (e.g., inversion of the order of execution of two operations).

The identified requirements for the input architectural model are summarized in Table I, categorized into features that should be supported (FE), and constraints that need to be imposed (CO).

B. Selected UML Elements

By looking at requirements in Table I it is evident that most of them can be fulfilled using UML. Some extensions need however to be introduced, mostly for distinguishing the interaction kind of components, and for representing time. For our "surface" UML representation we use elements from:

- UML [10], to model components, ports, and composite structures (req. 1, 2, 4, 5, 6, 7, 8, 9, 10, 11);

- fUML [11] (in its ALF [12] representation), to specify the behavior associated to operations (req. 12, 13, 14, 15);
- MARTE [13], to specify the periodicity of execution of operations, and to distinguish ports with different interaction kind (req. 2, 3, 4, 5, 7, 8, 9, 16, 17, 18).

Components, ports, and composite structures are modeled using the classical UML elements. The distinction between the "dataflow" and "function call" kinds of interactions is available in the MARTE "Generic Component Model" (GCM) package.

Ports specifying an interaction of "dataflow" kind can be modeled as MARTE «FlowPort» elements [13]. The adopted semantics is an adaptation of the "pull" semantics of non-behavioral FlowPorts [13]: *"data received on the port will be stored on the property targeted by the delegation connector, and replace any value contained in the property"*. In our case, we don't use delegation connectors to properties; conversely, we assume that an internal property having the same name exists for any input FlowPort of the component.

Similarly, ports describing "function call" interactions can be modeled as MARTE «ClientServerPort» elements [13]. ClientServerPort elements must be typed by (i.e., associated with) *Interface* elements. The interface associated with the port specifies the operations that can be called on the port. A "provided" ClientServerPort specifies operations that are offered by the component, i.e., that are implemented by that component. A "required" ClientServerPorts specifies operations that are needed by the component to perform its functionality, i.e., the operations that the component needs to invoke.

We adopt the semantics of non-behavioral ClientServerPort [13]. If no delegation connector exists, and the classifier of the receiving instance directly realizes the behavioral feature, the reception of the message directly triggers a call to this behavioral feature. If instead a delegation link exists, the received message follows one of the available delegation connectors and the message is handled by the delegation target.

With respect to requirements in Table I, the main missing information is i) the execution times of component operations, and ii) the specification of periodic execution of operations that are triggered by the execution environment. Such properties need to be specified using ad-hoc stereotypes. One possibility is to use stereotypes from MARTE: the *«ResourceUsage»* stereotype, with its *execTime* attribute, can be used to attach information about their execution time to component operations; similarly, the *PeriodicPattern* TupleType can be used to specify information about the periodic execution of operations [13].

C. Model-Transformation Requirements

Before actually describing our model-transformation approach, we briefly discuss the requirements that the code generation approach should implement (Table II). Requirements are grouped in different categories: *Input* (IO) and *Output* (OU), defining the inputs and outputs of the transformation procedure, and *Procedure* (PR), defining requirements for the transformation procedure itself. Basically, such requirements state that: i) the algorithm shall take as input a model with the characteristics specified in Section IV.A; ii) the algorithm shall produce an executable ALF model; and iii) the produced ALF model shall correctly represent the behavior of the component-based model received as input.

TABLE II. REQUIREMENTS OF MODEL-TRANSFORMATION PROCEDURE.

ID	Type	Description
1	IN	Transformation shall have as input the UML model with the feature and the constraint that are specified in Section IV.A.
2	IN	Transformation shall have as input the ALF implementation of the operations associated with the components.
3	OU	Transformation shall create a set of ALF source files, including the "main" activity ALF that describes the behavior of the entire system.
4	PR	Transformation shall define new components extracting information by the component model.
5	PR	Transformation shall instantiate the component and generate "connections" between them, extracting this information by the component model.
6	PR	Transformation shall handle timing, establishing the instant of time at which the operation is executed on the basis of the period, and of the phase of each operation.
7	PR	Transformation shall reproduce the behaviour of the system taking into account the execution time of each operation.

D. Model-Transformation Algorithm

The UML+MARTE+ALF representation described in the previous section is our instantiation of the "Surface UML" model mentioned in [11]. Our aim is to generate from it, via model-transformation, an ALF-only model (which is therefore executable) to be used as support to SW FMEA.

It is important to recall that the only structural elements available in fUML are UML Classes, i.e., concepts like *Component*, *Port,* and *InstanceSpecification* are not available in fUML. For this reason, each component in the source component-based model shall be mapped to *Class* elements, and their *Ports* need to be mapped to *properties.* Actually, due to some restrictions imposed by fUML, each component *instance* in the source model needs to be mapped to a different fUML *Class* in the target model. With respect to this implementation choice, it should be noted that the ALF-only model will be i) automatically generated, and ii) used only for the purpose of the simulation. Therefore, here we are mainly concerned in keeping the model-transformation algorithm simple.

The executable ALF model generated by our approach is actually a *collection* of ALF files, including a set of ALF source code files corresponding to the different components of the software architecture, and a "main" ALF file which serves as the entry point of the simulation. The model-transformation consists of various phases, which are detailed in the following.

To better understand the methodology, we consider the simple working example shown in Fig. 3, to which we will refer in each of the transformation steps listed below. The model in the example contains three component instances, "comp1", "comp2", and "comp3". *comp1* and *comp2* are connected through DataFlow ports, with datatype Integer, and direction from *comp1* towards *comp2*. Conversely, *comp1* and

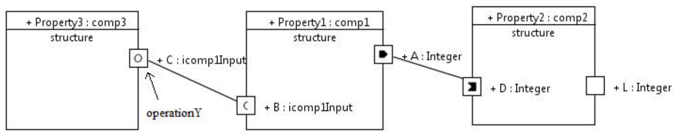

Fig. 3 Example model to support the description of the model-transformation.

comp3 are connected through ClientServerPort ports; in particular, *comp3* provides the interface "icomp1Input", which is instead required by *comp1*. Such interface contains the operation *operationY*.

1) *Mapping of Components*

The first phase consists in creating the structural part of the software architecture in the ALF-only representation. This is accomplished as follows. For each instance of an *atomic component* in the system, i.e., a component for which the internal structure has not been detailed, an ALF public *Class* is created. Furthermore, any *property* of the component instance is mapped as an ALF *private* property (i.e., variable) of the corresponding class. Each operation defined by the component from which the instance originates is mapped to an activity (i.e., method) of the generated ALF class.

2) *Mapping of Operations*

Operations defined on components, and their ALF bodies, need to be projected in the ALF-only model. In this first stage, the operations associated with each component are simply copied as public operations of ALF classes originating from the component.

3) *Mapping of FlowPorts*

The subsequent step involves the mapping of component *ports*. We first address FlowPorts, which have a simpler representation. The direction of a FlowPort may be *output*, if it emits data from the involved component, or *input*, if it receives data from other components. For each FlowPort of the component with an *output* dataflow, a *public* property of the same data type is created in the ALF class originated from the component instance. The value of this variable, which can be read from other ALF classes (since it is public), will represent the current value available on the corresponding FlowPort.

With reference to the example of Fig. 3, processing the FlowPort A of comp1, results in the following ALF code:

```
public class comp1 {
    public flowportA_value: Integer;
}
```

FlowPorts with an *input* dataflow are handled in a different way. For each of them, a *public* property is created in the class corresponding to the component instance. This property will be typed with the ALF Class originated from the component connected on the other side of the port. At runtime it will actually contain a reference to the component instance connected on the other side of the port.

4) *Mapping of ClientServerPorts*

ClientServerPort elements may have a *provided* or *required* direction. For ClientServerPort with direction *provided*, and for each operation defined on the associated interface, a corresponding *public* operation should be already defined on the class associated with the component, by step #2.

ClientServerPort with direction *required* are handled in a similar way as input FlowPorts: for each of them, a reference to the component instance connected on the other end of the link is added to the ALF Class. At runtime, this property will contain a reference to the component instance connected on the other side of the port.

With reference to Fig. 3, after processing ClientServerPort B of component *comp1*, the ALF Class corresponding to the component is modified as follows:

```
public class comp1 {
    public flowportA_value: Integer;
    public compOnB: comp3;
}
```

5) *Mapping of Connectors*

Once all the component instances and their ports have been projected into an ALF representation, connectors connecting each other component instances can be processed as well. In this step instances of ALF classes are created, and then, based on connectors exiting in the UML model, references to correct components are set in variables corresponding to ports.

Considering the example of Fig. 3, the following ALF code will be added to the "main" function of the generated ALF executable code.

```
comp1_inst = new comp1();
comp2_inst = new comp2();
comp3_inst = new comp3();
comp1_inst.compOnB = comp3_inst;
comp2_inst.compOnD = comp1_inst;
```

The interpretation of this code is as follows. "comp1", "comp2", and "comp3" are the three ALF classes corresponding to the different component instances. The first three lines instantiate the ALF classes, i.e., the component instances are added to the simulation. The fourth line establishes the connection, in the simulated environment, between components *comp1* and *comp3*, by setting the variable *compOnB* to a reference of *comp3_inst*. In this way all the operations calls that are performed on *compOnB* are actually operation calls on *comp3_inst*. Similarly, the last line establishes the connection between components *comp2* and *comp1*.

6) *Rewriting of operations bodies*

In the component-based model, the bodies of operations, specified in ALF, were constrained to refer to elements belonging to the component only (see Section IV.A). The elements that could be referenced included also FlowPorts and ClientServerPorts; in particular, FlowPorts could be used as variables, i.e., they can be used either in the left hand or in the right hand of an assignment expression. Since ports are not in the executable ALF model, bodies of operations specified in the input model need to be rewritten, resolving references to ports based on the actual connections between components.

References to *output FlowPorts* are simply replaced by the corresponding property (variable): any reading or writing of the corresponding port translates in a reading or writing of the corresponding variable. Concerning *input FlowPorts*, they are replaced by a reference to the corresponding property in the component connected on the other end of the port. The property can be referenced thanks to the connection established in step #5. Finally, references to operations called on *required ClientServerPorts* are replaced by a reference to the corresponding operation in the component connected on the other end of the port. Also in this case, the connected component can be referenced thanks to the links established in step #5.

With reference to the example of Fig. 3, let us assume that *comp1* has an operation "run()", with the following ALF body:

```
public run() {
    A = B.operationY() * 10;
}
```

The operation body is then rewritten as:

```
public run() {
    this.flowportA_value = this.compOnB.operationY() * 10;
}
```

7) Representation of Time

It should be stressed that the concept of time is not supported in ALF. This problem has been recognized in e.g., [27], which proposes an "explicit" scheduler for fUML execution.

Our current practical approach for handling time is a simplification of such execution model, where time is simulated directly in the ALF text using an *indexed loop*, where each loop represents the elapsing of one unit of time (i.e., a "tick"). As described in Sections IV.A and IV.B, our notion of time is characterized by three parameters: the *period*, *phase* and (optionally) the *duration* of operations.

The period and phase parameters of a periodically executed operation are simulated by actually calling such operation only when the required timing conditions hold. For example, an operation that is called periodically every 100 time units will be executed only when the current time (i.e., index) modulo 100 is zero. Concerning the *duration* of an operation, in principle it could be simulated by simply increasing the loop index after its execution. Unfortunately, constraints imposed in the fUML specification do not allow loop variables to be directly modified. A workaround consists in introducing a support variable, which holds the number of "ticks" that should be skipped after the operations has been executed.

Consider the system in Fig. 3 and assuming that components 1 and 2 both have an operation named "run()" which is executed periodically. "1.run()" is scheduled to be executed every 100 time units, while 2.run() should be executed every 500, with an initial delay of 200. Suppose that the "1.run()" operation requires 10 time units, while "2.run()" requires 20 time units. The ALF "main" activity that simulates the system from time 0 until time 1000 would include the following code:

```
for ( j in 1..1000 ) {
    if ( k == 0 ) {
        if( j % 100 == 0 ) {
            comp1_inst.run();
            k = 10;
        }
        if( (j >= 200) & ((j – 200) % 500 == 0)) {
            comp2_inst.run();
            k = 20;
        }
    }else{ % Do nothing: just loop on more time }
}
```

V. Current Prototype and Next Steps

A prototype of the approach described in this paper is being realized within the Eclipse platform, allowing us to verify the feasibility of the project and test it on real use-cases in the automotive domain. Besides the model-transformation described in this paper, work is ongoing for realizing other parts of the workflow as well. Details on these individual activities are discussed in the following.

A. Model Creation and Editing

One of the first activities in realizing our workflow has been the identification of a suitable UML model and diagram editor capable of creating models with the characteristics described in Section IV.B. The editor that we are using for our prototype is the CHESS Editor [20], a customized Eclipse-based editor developed within the CHESS project [20], [17]. The CHESS editor is capable of editing CHESS ML, UML and MARTE models, and it is thus also able to create models suitable for our approach. Furthermore, it also contains an ALF editor with syntax highlighting, which can be used to provide the ALF implementation to operation bodies.

B. Model Execution

To build our prototype we investigated some of the available tools for the execution of fUML models. The "official" fUML simulator is the "Foundational UML Reference Implementation" [18]. Unfortunately, the tool performs strict checks on the format of the input UML model, which required cumbersome manual modifications to the XMI file. For this reason it was not selected as a suitable candidate. Another fUML-enabled tool is "AMJUSE", which is however a plugin to a commercial tool. Another candidate was the Moliz [9] tool introduced in Section II. Among its characteristics, Moliz was able to use as input "regular" UML model produced by the Papyrus diagram editor [21], and was also able to handle UML models with profiles applied to them. Unfortunately, Moliz only supports the fUML representation, which – for our purposes – is much less convenient than the text-based ALF one.

We finally selected the "ALF Reference Implementation" [19], an open-source Java-based implementation of the ALF standard, capable of interpreting and simulating a collection of textual ALF source code files. In our experiments the ALF "main" activity obtained from the transformation in Section IV.D was successfully executed by such tool. The tool also provides facilities to write and read text files, which can be used to read input values and write simulation traces. In our prototype, such feature is used to provide a nominal *workload* to the system, and to produce traces of specific observation points over time.

Output traces of "fault-free" executions are recorded, together with the associated workload. Two main challenges we identified in this aspect are: i) selecting the proper size of the simulation step for providing inputs and collecting outputs; and ii) identifying the relevant set of events when comparingthe "golden" and "faulty" traces. These aspects are currently being investigated and will be addressed in next refinement steps.

C. Fault Injection

Concerning fault-injection, we have identified different levels at which it could be performed in such a workflow. First of all, fault-injection can be performed directly on the ALF-only model generated by the transformation, or as part of the model-transformation algorithm itself. The latter seems to be more convenient, since information from the component-based model could be used to guide the application of faults.

Also, we identified two levels in the definition of fault models: the *component* level, and the *code* level. In the former option, faults are injected at component boundaries, i.e., components' ports. In the latter, faults are injected directly in the ALF specification of operation bodies. Since we are most interested in the effects of propagation of faults, rather than on their *causes*, we inject faults on component boundaries. Our effort is currently focused on the definition of the fault library, based on the failure classification adopted, among others, in [3]: incorrect value, omission and commission, early and late. For example, in our context, an omission fault is injected by removing a call to a function through a ClientServerPort. Our future work will therefore focus on fault-injection aspects, and apply the approach to an industrial case study.

VI. Concluding Remarks

The importance of safety analysis of software architectures is growing. Such analysis should be carried out *early enough* to detect possible insufficiencies in the safety mechanisms, but at the same time the design process has to *advance enough* to ensure a proper correlation between the model used for SW FMEA and the later actual implementation.

The approach proposed in this paper is based on model-execution and fault-injection. This paper detailed on the process of obtaining an executable model from a richer, general purpose, component-based UML representation. We first specified which information is necessary to build an executable model from a component-based description of the SW architecture. Then, we identified a set of useful UML and MARTE constructs that can be used to actually specify such information. Finally, we described a model-transformation that actually generates a fUML executable model. The obtained executable model is at the basis of our approach for performing SW FMEA via fault-injection.

A prototype centered around the Eclipse platform already implements part of the workflow. Ongoing work is aiming at integrating the prototype into the CHESS-CONCERTO framework [20], a cross-domain system design framework including techniques for addressing of non-functional concerns.

Acknowledgment

This work has been partially supported by the CECRIS, "CErtification of CRItical Systems", 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, and by the MS-VIVA project, funded by the Tuscany Region within the framework POR CReO FESR.

References

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

[2] Ward, P. and Armstrong, J.: "Behavioral fault simulation in VHDL" in Proc. Design Automation Conference, pp. 586-593, (1990).

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

[4] Papadopoulos, Y. et al. "Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure" Reliability Engineering & System Safety, 71, 229–247 (2001).

[5] Hong, Z., Lili X., "Application of Software Safety Analysis Using Event-B," Seventh International Conference on Software Security and Reliability Companion, 18-20, pp. 137-144 (2013).

[6] Dittel, T., Aryus, H.-J., "How to "survive" a safety case according to ISO 26262. In: Com-puter Safety, Reliability, and Security," LNCS vol. 6351, Springer, pp. 97-111 (2010).

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

[8] Schubotz, H., "Experience with ISO WD 26262 in Automotive Safety Projects," SAE Tech Paper (2008).

[9] 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).

[10] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August. 2011.

[11] Object Management Group, "Semantics of a Foundational Subset for Executable UML Models (fUML)", Version 1.1, August 2013.

[12] Object Management Group, "Action Language for Foundational UML (Alf)," Version 1.0.1, October 2013.

[13] Object Management Group, "UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems," Version 1.1, (2011).

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

[15] Madeira, H., Vieira, M., "On the Emulation of Software Faults by Software Fault Injection," Dependable Systems and Networks (DSN), pp. 417-426, (2000).

[16] Svenningsson, R., Törngren, M., "Model-Implemented Fault Injection for Hardware Fault Simulation," Workshop on Model-Driven Engineering, Verification, and Validation, pp. 31-36, (2010).

[17] Montecchi, L., Lollini, P., and Bondavalli, A., "Towards a MDE Transformation Workflow for Dependability Analysis," In Proc. of the 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2011), pp.157-166 (2011).

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

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

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

[21] Gérard, S. at al. "Papyrus: A UML2 Tool for Domain-Specific Language Modeling", In *Model-Based Engineering of Embedded Real-Time Systems*, Springer, 6100, 361-368 (2001).

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

[23] Pataricza, A. 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.

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

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

[26] Ciccozzi, F., Cicchetti, A., Sjödin, "Towards Translational Execution of Action Language for Foundational UML", 39th Euromicro Conference Series on Software Engineering and Advanced Applications, 2013.

[27] Benyahia, A., et al. "Extending the Standard Execution Model of UML for Real-Time Systems" DIPES/BICC, Springer, 43-54, 2010.

[28] C. Hofmeister, R.L. Nord, D. Soni. "Describing Software Architecture with UML", In *Software Architecture*, IFIP, Vol. 12, pp 145-159 (1999).

[29] Z. Micskei and H. Waeselynck: "The many meanings of UML 2 Sequence Diagrams: a survey", Software and Systems Modeling, Vol. 10, Num. 4, pp. 489-514, Springer (2011).