A Model-driven Dependability Analysis Method for Component-based Architectures

Barbara Gallina, Muhammad Atif Javed, Faiz UL Muram and Sasikumar Punnekkat
MRTC, School of Innovation, Design and Engineering,
Mälardalen University,
Västerås, Sweden
{barbara.gallina, sasikumar.punnekkat}@mdh.se
{mjd09002, fmm10001}@student.mdh.se

***Abstract*—Critical distributed real-time embedded component-based systems must be dependable and thus be able to avoid unacceptable failures. To efficiently evaluate the dependability of the assembly obtained by selecting and composing components, well-integrated and tool-supported techniques are needed. Currently, no satisfying tool-supported technique fully integrated in the development life-cycle exists. To overcome this limitation, we propose CHESS-FLA, which is a model-driven failure logic analysis method. CHESS-FLA allows designers to: model the nominal as well as the failure behaviour of their architectures; automatically perform dependability analysis through a model transformation; and, finally, ease the interpretation of the analysis results through back-propagation onto the original architectural model. CHESS-FLA is part of an industrial quality tool-set for the functional and extra-functional development of high integrity embedded component-based systems, developed within the EU-ARTEMIS funded CHESS project. Finally, we present a case study taken from the telecommunication domain to illustrate and assess the proposed method.**

Keywords- Component-based Architectures, Dependability, Failure Logic Analysis, Model-driven engineering.

I. INTRODUCTION

Component-based architectures are the result of the selection and composition of reusable components. The process of selecting and composing components is evolutionary and its ultimate goal is the achievement of a result that optimizes the quality and business trade-offs. In case of critical systems, quality and, more specifically, dependability is fundamental and therefore approaches for the analysis of causation paths regarding dependability threats (i.e. failures) are necessary to plan adequate counter-measures. At the same time stringent requirements with respect to time to market necessitate the availability of tool-supported analysis. The architectural level represents an abstraction level that permits designers to understand and analyse the needed trade-offs. Therefore, evolution in terms of architecture restructuring (e.g. reconfiguration, components replacement, etc.) is crucial at this level. To support evolution towards the achievement of dependable architectures, compositional and automatic techniques to analyse the failure behaviour at system level are necessary. Moreover, these techniques have to be integrated within a development process allowing for separation of concerns.

The CHESS project [2] aims at providing a tool-supported model driven engineering framework for the assembly of high integrity embedded component-based systems. For such, the tool-set developed within CHESS permits users to address functional as well as extra-functional concerns. In the framework of CHESS, we propose CHESS-FLA which is a model-driven failure logic analysis method fully integrated within the CHESS tool-set. CHESS-FLA is specifically conceived to be used during the design phase and it supports designers and dependability experts in several activities. More specifically, designers have at disposal means to model the nominal as well as the failure behaviour of their software architectures. Moreover, they can automatically perform dependability analysis, more specifically failure logic analysis. Models enriched with the required information for the specific analysis are transformed into the input format required by the analysis engine. Finally, thanks to the back-propagation of the analysis results onto the original architectural model, designers may interpret the results more easily.

The main novelty of CHESS-FLA consists in the integration of a promising failure logic analysis approach within an industrial-quality and model-driven development framework that targets the functional as well as extra-functional assembly of component-based systems. The availability of such integration allows designers as well as safety experts to reduce time effort thanks to the automation of the analysis as well as thanks to the possibility of using the same functional model. By having at disposal such tool support, designers may repeat the analysis more frequently during the architectural evolution and thus increase the chances of achieving accurately planned dependable architectures. Moreover, since CHESS-FLA also offer methodological guidelines, designers and dependability experts are supported for the interpretation of the analysis results as well as for the selection of mitigation strategies.

The remainder of this paper is organized as follows. Section II recalls essential background information. Section III presents the proposed model-driven dependability analysis method. Section IV presents a case-study. Section V discusses related work. Finally, Section VI provides conclusions and directions for future work.

II. BACKGROUND

In this section, we recall the failure logic modelling and analysis approaches on which we base our work and we give

an overview of the CHESS tool-set, in which we integrate our work.

A. Failure Logic Modelling and Analysis Approaches

Failure logic modelling and analysis approaches introduce the possibility to unify as well as automatize existing traditional dependability analysis approaches (e.g. Fault Tree Analysis and Failure Mode, Effects and Criticality Analysis) [12]. Several failure logic modelling and analysis approaches are available in the literature. Failure Propagation and Transformation Notation (FPTN) [5] and Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) [16] were introduced first. These approaches offer a hierarchical and semi-graphical notation which is not completely defined in the first approach and more mature in the second one. Besides FPTN and HiP-HOPS, other failure logic modelling approaches have emerged. As discussed in [9], these approaches present an important limitation: they do not support the analysis of component-based systems that exhibit cycles (cyclic data-and control-flow structures). Failure Propagation Transformation Calculus (FPTC) [17] overcomes this limitation. Since many real-world control systems contain closed feedback loops, FPTC overcomes a significant limitation.

Moreover, FPTC improves its predecessors (FPTN and HiP-HOPS) with respect to other dimensions: formality and usability. FPTC, indeed, offers: a formal syntax and a formal semantics; means to reduce the specification burden thanks for instance to the availability of the wildcard symbol, which permits users to indicate any behaviour.

From a tool-support point of view, currently, an implementation has been provided within the Epsilon model management platform [15]. Despite its relevance in terms of technology independence as well as independence from UML-based languages, its current status cannot be considered adequate to be integrated within an industrial-quality environment. The modelling capabilities offered by this implementation are very basic. Architectures can be modelled only through blocks and lines. No support for modelling interfaces is available. Moreover, layered architectures cannot be modelled.

For the above summarized advantages offered by FPTC, our method introduced in Section III, builds upon it. More specifically it builds on FI^{4}FA (Formalism for Incompletion, Inconsistency, Interference and Impermanence Failures Analysis) [7], which is an extension of FPTC to take into consideration Incompletion, Inconsistency, Interference and Impermanence Failures and their corresponding counter-measures. Therefore, in this subsection, we recall the FI^{4}FA approach by distinguishing its similarities and differences with respect to its predecessor.

FI^{4}FA, similarly to FPTC, is a technique for automatically calculating the failure behaviour of an entire system from the failure behaviour of its individual components. The failure behaviour of the individual components, established by studying the components in isolation, is expressed by a set of logical expressions (FI^{4}FA rules) that relate output failures (occurring on output ports) to combinations of input failures (occurring on input ports). Output failures due to internal faults are implicitly specified by relating output failures to normal behaviour in input. Input failures whose corresponding output behaviour is not specified are propagated as they are. Input failures are assumed to be propagated or transformed deterministically. Thus, to a failure in input is associated only one output behaviour.

The inter-connected components are considered as a token (failure/no-failure)-passing network. To determine the behaviour at system level, it is necessary to consider the set of all possible behaviours (failures and or normal behaviour) that can be propagated along a connection (called tokenset). More specifically, the behaviour at system level is obtained through a fixed-point calculation that calculates the maximal tokenset on any connection in the network. Therefore the system structure is a fundamental piece of information to be able to identify the propagation paths. For further details concerning how the analysis is carried out and under which assumptions the reader may refer to [7].

FI^{4}FA differs from FPTC because it offers two additional features. The first feature consists of the possibility to carry out a more fine-grained analysis of the failure behaviour. This feature is obtained by introducing additional failure types, which enable the analysis of Incompletion, Inconsistency, Interference and Impermanence (I^4) failures. The second feature consists of the possibility to model and analyse the mitigation behaviour. This feature is obtained by introducing mitigation types, seen as specialization of normal behaviour. The mitigation types as well as the failure types that are introduced in FI^{4}FA can be also interpreted as tokens in a token-passing network and, therefore, the behaviour at system level is calculated in the same way as in FPTC.

Initially, the necessity of modelling and analysing I^4 failures emerged in the framework of transaction and component-based systems to offer means to better discriminate failures and thus tune the desired combination of ACID (Atomicity, Consistency, Isolation and Durability) properties. The gained awareness concerning the conceptual breadth of the term “transactional” [6] (which should not be interpreted only within the database-related semantic domain but should be associated to the ongoing research to address dependability, more specifically reliability, in modern systems which can be characterized by highly distribution, autonomy, heterogeneity, cooperation needs) make us believe that FI^{4}FA can be used more extensively. Moreover, as we pointed out in [8], if we consider the concept “transactional” and not how to implement it, even a packet transmission in networked systems can be seen as a transaction. Similar to a transaction, indeed, a packet transmission is supposed to make the system transit from a consistent state to another consistent state.

Both FPTC and FI^{4}FA permit users to describe purely event-based failure behaviour, input-output behavioural description without any information regarding the state. In case of component-based systems built from components from the shelf, this description is the only possible one since the internal behaviour remains unknown. FI^{4}FA represents a promising failure logic analysis technique and thus has been selected to be integrated within the CHESS framework.

B. Overview of the CHESS Tool-set

The CHESS tool-set, soon available in open-source, constitutes the main work-product of the CHESS project [2]. This tool-set, conceived as a set of Eclipse plug-ins, is a model-driven engineering infrastructure that permits high integrity component-based embedded systems to be assembled. More specifically, these systems are obtained through a series of model-transformations. The development team first of all has to provide the platform independent model (PIM) using the modelling language CHESS-ML. Then, the team has to choose a platform and enrich the PIM with deployment information to be able to automatically generate the platform-specific model (PSM). Finally, from the PSM, code can be generated automatically. Besides model transformations, used to achieve the final component-based implementation, additional model transformations and back-propagations allow models to be analysed to guarantee that the final implementation satisfies required extra-functional properties (e.g. dependability).

III. CHESS-FLA

CHESS-FLA is a tool-supported model-driven dependability analysis method to be used during the early as well as late stages of the design phase of evolving component-based architectures. CHESS-FLA stands for Failure Logic Analysis within the CHESS framework. Following a model-driven approach, CHESS-FLA integrates within the CHESS framework FI^{4}FA and thus contributes to the composition with guarantees of high-integrity software components. In addition, CHESS-FLA contributes in reducing time-effort. To achieve CHESS-FLA, we have performed the following steps:

1. identification of the dependability concepts needed to support the analysis method;
2. inclusion of the identified concepts within CHESS-ML;
3. extension of the analysis engine;
4. integration of the analysis engine according to a model-driven approach. This step has required the conception of transformations;
5. provision of back-propagations to ease the readability of the analysis results;
6. provision of a process and guidelines to support designers during architecture evolution. This process and guidelines suggest who should do what and in which way (i.e. specific counter-measures to address specific failures).

These steps are briefly discussed in the following subsections. In particular, Section III.A discusses step1; Section III.B and Section III.C discuss step2; Section III.D discusses step 3, Section III.E discusses steps4-5; finally, Section III.F discusses the last step.

A. Dependability concepts identification

As mentioned in Section II.A, to be able to perform the analysis, first of all the analyst must have at disposal the concepts needed to model the structure of the system. The port-to-port interconnection among components is indeed crucial information to trace the propagation paths. Concepts needed to model the dependability threats and the causality chain that relate them are also essential to be able to model the input-output behaviour at component level and thus model how potential threats received in input are propagated/transformed on the output. In case of normal behaviour achieved through the application of counter-measures, it is necessary to have at disposal the concepts needed to model them. Finally, the concepts to model the analysis context are also needed.

These concepts share the dependability conceptual framework achieved as a joint work in the framework of the CHESS project and initially introduced in [14].

B. CHESS-ML

CHESS-ML is a cross-domain modelling language conceived in the framework of the CHESS project. This language is defined as a collection-extension of subsets of standard OMG languages (UML, MARTE, SysML) and its objective is to allow users to define platform independent as well as platform specific models. Moreover, CHESS-ML supports the definition of various analysis models (e.g. dependability models) within specific views. The CHESS-ML constructs which are related to the FI^{4}FA analysis are briefly introduced below:

Constructs for the static structure

Components are the basic blocks of the system. Components can be classified in various ways: on the basis of their nature (software or hardware); on the basis of their structure (atomic or composite); on the basis the refinement step (platform-independent or platform-specific).

Ports. Components have ports which can be classified as client/server (used for software components) or flow (used for hardware components) ports.

Interface. Software components have interfaces through which they provide or require services. To an interface is associated a set of operations. On the basis of the kind of interface and the direction of the operations the type (input or output) of the port of the component can be identified.

Constructs for modelling failure data

FI4FA stereotype is introduced for decorating a component implementation with set of propagation and transformation rule(s). More specifically, the attribute called *fi4fa* is used to provide the rules, according to the syntax given in Section III.C This stereotype can be directly applied on a component implementation or alternatively on a comment attached to the component implementation itself.

FI4FASpecifation stereotype is introduced to provide failure data on input and output ports. More specifically, the attribute called *failure* is used to model the desired type of failure(s). It must be noted that in case of no failure, the attribute failure is associated with the value “noFailure”.

ACIDAvoidable stereotype is introduced to specify the I^4 failures.

ACIDMitigation stereotype is used to specify mitigation means specifically adequate in addressing I^4 failures.

Constructs for the analysis context

FI4FAAnalysis stereotype is introduced to define the analysis context. This stereotype is used to extract the

information regarding target platform instance specification for executing FI^{4}FA analysis. FI4FAAnalysis stereotype is specified in DependabilityAnalysisView.

C. Syntax adaptation

As explained in Section III.B, within CHESS-FLA, the dependability experts provide behavioural specifications (FI^{4}FA rules) for individual components by setting the attribute *fi4fa*.

The syntax for the FI^{4}FA rules, introduced in [7], has been slightly modified to be used in CHESS-FLA. In particular, the possibility to explicitly link the behaviour to a specific port name is given. To allow readers to understand the behavioural specifications provided in Section IV.A, below are given the syntactical rules that must be used to specify the failure behaviour of the individual components to perform the FI^{4}FA analysis:

behaviour = expression +
expression = LHS '→' RHS
LHS = portname'.' bL | portname '.' bL (',' portname '.' bL) +
RHS = portname'.' bR | portname '.' bR (',' portname '.' bR) +
failure = 'early'| 'late' | 'commission' | 'omission' | 'valueSubtle' | 'valueCoarse'
bL = 'wildcard' | bR
bR = 'noFailure' | 'noFailure'.ACIDMitigation | failure.ACIDAvoidable | '{' failure.ACIDAvoidable (',' failure.ACIDAvoidable) + '}'
ACIDavoidable=Aavoidable.Cavoidable.Iavoidable.Davoidable
Aavoidable= 'incompletion' | none | unspecified
Cavoidable = 'inconsistency' | none | unspecified
Iavoidable = 'interference' | none | unspecified
Davoidable = 'impermanence' | none | unspecified
ACIDmitigation=Amitig'.'Cmitig'.'Imitig'.'Dmitig
Amitig = 'all-or-nothing' | 'all-or-compensation' | 'none'
Cmitig = 'full-consistency' | 'range-violation' | 'none'
Imitig = 'serializable' | 'none'
Dmitig = 'no-loss' | 'partial-loss' | 'none'

Thus, an example of a compliant expression is:

C1_R1.noFailure→
C1_P1.valueCoarse.incompletion.none.none.none

For sake of clarity, it must be observed that FPTC rules can be obtained from FI^{4}FA rules by omitting the information concerning the mitigation and by using only unspecified ACID avoidable failures. The above rule should be read as follows: if the component C1 receives on its port R1 a normal behaviour, it generates on its output port P1 a coarse (i.e. clearly detectable) value failure related to incompletion.

It must be observed that the behaviour specified at the component implementation level in terms of failure logic, is abstract and can be reused independently of the application context and domain. At the component instance level, however, in case the context or domain introduces additional behavioural information, the failure logic specifications can be refined.

D. Extension of the analysis engine

FLA is an analysis engine implemented in JAVA to perform failure logic analysis (FI^{4}FA). FLA takes as input an XML file that contains information necessary to perform the analysis and generates as output an XML file that contains the results of the analysis. This engine is the result of a significant extension of the original analysis engine (limited to FPTC) that was presented in [11].
The XML-based input-output format has been conceived to make it possible to reuse the engine in other contexts (i.e. to integrate it in other tools).
The extension is not only limited to FI^{4}FA support. FLA enables hierarchical analysis. To perform failure logic analysis of hierarchical systems, two possible ways are available. The first way is fully automatic and permits the hierarchical calculus of the failure behaviour. The second way is semi-automatic and requires the user to perform the analysis step by step from the bottom to the top (from sub-systems to system).

In case of heterogeneous systems, a step by step analysis permits users to start with domain-specific and context-specific failure analysis at homogeneous levels and then move to a more abstract analysis after having considered refinement/abstraction rules to be able to manage the terminological difference that usually exists in naming failure types (the ways in which failures manifest themselves).

By having abstract failure types such as (value, timing, provision failures) at the non-homogeneous level, dependability experts can succeed in writing propagation and transformation rules that enable the automatic calculation of the propagation and transformation paths among components. Moreover, the annotated models become more readable. At the same time, having at disposal refinement rules to move from abstract to context-specific or domain specific rules, gives the opportunity to perform a more fine-grained analysis and thus identify more precise counter-measures to mitigate unacceptable failures.

This step-by-step analysis thus contributes in solving the issues concerning the applicability of failure logic analysis techniques discussed in [13]. For example, in the networking domain, typical failure modes are: duplication, collision, bit flip, etc., and in the railway domain typical failure modes related to a track are: surface fatigue, forced fracture, etc. It is clear that to model failure propagation within systems composed of non-homogeneous sub-systems, failures modes must be more abstract to allow the establishment of causation paths. More abstractly, a bit flip can be specified as coarse value failure since the value of the bit differs from the expected value in a detectable way. Similarly, a surface fatigue can be specified as coarse value failure. In the context of integrated behavioural analysis of railway systems composed of tracks, communication and control systems, such an abstract specification is essential to infer potential failure propagation paths among heterogeneous components.

E. CHESS-FLA implementation

To integrate FLA within CHESS and achieve CHESS-FLA, we have implemented the following steps, which are illustrated in Figure 1:

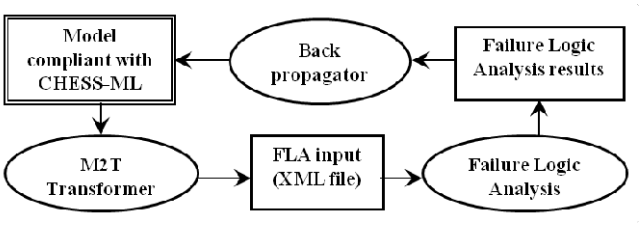

Figure 1. CHESS-FLA workflow

The first step consists in extracting the required dependability information from the CHESS-ML model to be analysed. As previously explained, such information concerns composite components and their related failure data in input, atomic components and their related behavioural propagation and transformation rules through input and output ports, and, finally, connections among components. This information, through a model-to-text transformation, is extracted and transformed into an XML document. The Model-To-Text (M2T) transformation is specified in the framework of Acceleo [1], which is an implementation of the OMG standard (MTL) and thus allows text to be generated from models. The M2T transformation consists of a template (.mtl file) containing transformation rules (expressions specified over meta-model entities with queries) for values to be selected and extracted from models. These values are converted into isolated text by using string manipulation library and enclosed in the output transformation file.

The second step consists in reading the information contained in the XML document generated in the previous step, storing it into data structures and performing the analysis based on that information.

The last step consists in enriching the initial CHESS-ML model by back propagating the analysis results onto it and thus increasing the readability of the analysis results. This back-propagation is implemented in Java.

F. CHESS-FLA process

The process to be followed for applying the failure logic analysis by using CHESS-FLA is illustrated in Figure 2.

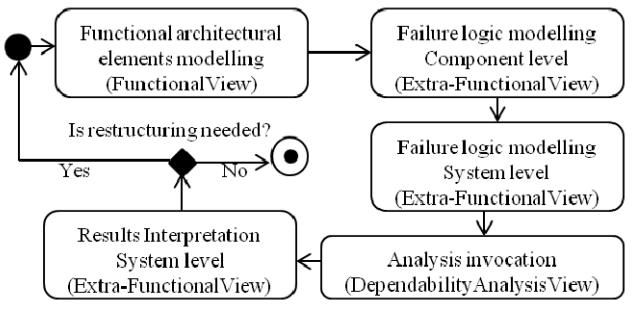

Figure 2. Activity diagram showing the CHESS-FLA process

The following enumeration briefly explains the steps of the process and its corresponding work-products (the interested reader may find further details in the CHESS project deliverables):

1-The first step consists in modelling: component types, component implementations and composite components by inter-connecting instances of reusable atomic component implementations or other composite component instances. This step is performed in the FunctionalView (within the ComponentView) by the designer. The work-product of this step consists of functional models regarding component implementations.

2-The second step consists in decorating the component implementations with the extra-functional properties needed for the analysis purposes. In case of FI^{4}FA analysis, this decoration entails the specification of FI^{4}FA rules. This step is performed in the Extra-FunctionalView (within the ComponentView) by the dependability expert. The work-product of this step consists of extra-functional models regarding component implementations.

3-The third step consists in decorating the software architecture to be analysed. In particular, the dependability expert must model the failure data that the software system may receive in input. The work-product of this step consists of extra-functional models regarding composite systems.

4-The fourth step consists of the selection of the analysis context and of the invocation of FI^{4}FA. This step is performed in the DependabilityAnalysisView and produces a model containing contextual information for the analysis.

5-Finally, after the back-propagation of the analysis results, dependability experts and designers can interpret the results and take appropriate design decisions. This step is performed in the Extra-FunctionalView. The work-product of this step consists of extra-functional models regarding composite systems after the back-propagation of the analysis results.

As general guidelines, in case the system behaves normally, it can be considered robust enough with respect to the behaviour considered in input. In contrast, in case the system exhibits unacceptable failures, counter-measures are needed. Since failure propagation paths are accessible and understandable thanks to the back-propagation, components that generate failures can be easily detected and therefore, if needed, can be combined with additional components that implements counter-measures (e.g. software redundancy exemplified by N-version programming). The introduction of new components or the replacement of old ones are examples of architecture restructuring strategies aimed at increasing architecture dependability. Guidelines supporting design decisions are available as CHESS project deliverables and build upon [20]. To restructure the architecture the process's steps have to be repeated. However, thanks to the reuse-based approach, changes are localized and require minor efforts.

IV. CASE-STUDY

CHESS-FLA has been applied to analyse rather complex (layered architectures having layers composed of five to ten components) systems such as the Asynchronous transfer mode Adaptation Layer type 2 (AAL2) system, documented in the CHESS deliverables. However, for space reasons and illustration purposes we choose to present in this section a simpler system. In particular, we apply CHESS-FLA to model and analyse the failure behaviour of a simple component-based system related to a packet transmission

protocol. The system is constituted of three chained single-input-single-output components representing respectively the source of the packet transmission (called Source), the intermediate component in charge of switching the packet to be transmitted (called Middle) and, finally, the destination of the transmission (called Destination).

The presentation of the application of CHESS-FLA assumes that a functional model already exists (work product of step1, as presented in Section III.E) and illustrates the remaining process 'steps.

A. *Individual components behaviour*

As explained in Section II.A, to perform FI^{4}FA, the behaviour of individual components must be studied in isolation. Since we have no possibility to study the components through direct testing; the behaviour is obtained by following a HAZOP (HAZard and OPerability analysis)-like approach. More specifically, each failure type available in FI^{4}FA is used as guideword to prompt possible failure behaviours of each component.

For brevity, the below-given rules do not specify the ports since Source, Middle and Destination are modelled as single input-single output components. However, when using CHESS-FLA, rules must be compliant with the syntax recalled in Section III.C.

Source

In case Source receives a normal behaviour, it generates a coarse value failure related to impermanence.

noFailure→valueCoarse.none.none.none.impermanence;

In case the Source receives a late failure in input, it transforms it into an omission failure related to incompletion.

late→omission.incompletion.none.none.none;

Middle

In case Middle receives a coarse value failure related to impermanence, it fully mitigates it by guaranteeing no loss.

valueCoarse.none.none.none.impermanence→ noFailure.none.none.none.no_loss;

In case Middle receives a normal behaviour, it propagates it.

noFailure→noFailure;

Destination

In case Destination receives a coarse value failure, it propagates it.

valueCoarse.none.none.none.none→ valueCoarse .none.none.none.none;

In case Destination receives a normal behaviour, it generates a coarse value failure related to incompletion.

noFailure→ valueCoarse.incompletion.none.none.none;

Figure 3 shows the decoration of the component implementation using Source as an example.

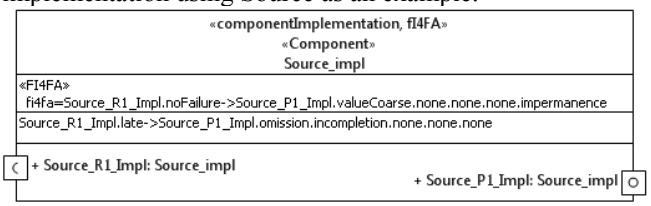

Figure 3. Decoration of Source with FI^{4}FA rules.

B. *FI^{4}FA Analysis*

In this subsection, we perform FI^{4}FA analysis on the component-based system that implements the toy packet transmission protocol. To evaluate the failure behaviour of this system, we specify a normal behaviour in input (*noFailure*), by adding a comment stereotyped with *FI4FASpecification* to the system's input port. Before the analysis only the behaviour in input is known. The model of the system after performing FI^{4}FA analysis is illustrated in Figure 4. As Figure shows, the system generates a coarse value failure related to incompletion on the output port.

This result is motivated as follows: Source is fed with normal behaviour and it generates a value coarse failure related to impermanence on its output port; Middle receives this failure on its input port and according to its first rule it provides on the output port a normal behaviour. Finally, Destination receives a normal behaviour and, according to its last rule, transforms it into a coarse value failure related to incompletion. As a result, this last failure is perceived on the output port of the system. Running the analysis and back-propagating its results take less than ten seconds (more precisely, 9.7 sec) for architectural models of this simplicity. More complex architectural models (i.e. 3-layered component-based architectures as in the case of the AAL2 system, mentioned at the beginning of the section) take less than twenty seconds (more precisely, 18.7 sec).

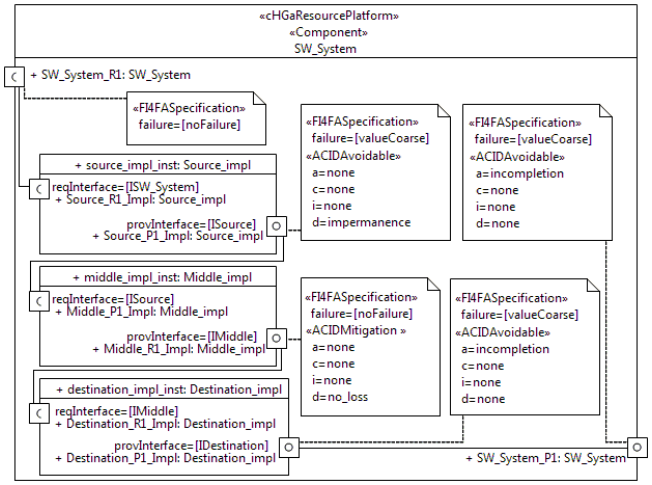

Figure 4. Back-propagation of the analysis results.

C. *Interpretation of the analysis results*

Once the analysis results are back-propagated onto the original model, components that generate failures (that behave like sources) can be identified. Similarly components that transform acceptable failures into unacceptable failures can be identified. As a counter-measure, designers and dependability experts may decide to replace components that exhibit an unacceptable behaviour, or they may decide to introduce additional components (software redundancy) to prevent failures from propagating beyond the top-system boundary or the specified sub-system boundary. Alternatively, designers may decide to introduce a mitigation means (implemented as a wrapper) at top-system level. With respect to the system analysed in Section

IV.B, the designer may decide to mitigate the coarse value failure related to incompletion by applying specific Atomicity-based mitigation means (e.g. packet reconstruction strategies). Alternatively, the designer may decide to apply more general mitigation means aimed at counteracting value failures, as discussed in [20].

It is crucial to stress that to each decision corresponds a restructuring of the architecture (an evolution), which can further be analysed. Designers and experts are, indeed, expected to repeat the process's steps until they reach a satisfactory work-product.

V. Related work

During the last two decades, several research works have contributed in advancing the state of the art in terms of modelling and analysis of failure behaviour of component-based architectures. State-based as well as stateless (purely event-based) formalisms and techniques have been proposed. In this section, we do not discuss all these works but rather limit us to a subset of stateless formalisms and techniques, more specifically, to FLA approaches. These approaches adopt an observational behaviour which, in our opinion and as pointed out in [19], is more suitable in case of component-based systems built from Commercial-Off-The-Shelf (COTS). Moreover, we focus our attention to those approaches that have been integrated, at least partially, within a tool-supported development approach.

In [5], authors point out the absence of an integrated toolset for safety analysis. According to the authors, the toolset should offer a common semantic framework to unify similar concepts used by different techniques or architectural views as well as a methodological support. To address the identified lack, authors provide a notation called Failure Propagation and Transformation Notation (FPTN). As mentioned in Section II.B, FPTN represents the first FLA approach that unifies FTA and FMECA analysis. Authors also discuss the possibility of generating automatically FPTN descriptions from design description; however, no concrete implementation has been provided.

In [19], authors explain how to model failure logic using CSP and thus benefiting from the validation and verification CSP tools. The possibility of generating automatically CSP models from annotated architectural models is only mentioned as potential future work. Thus, the analysis is supported but not integrated in a model-driven method.

In [10], authors propose a technique for the automatic generation of component fault trees (CFT) from component-level failure annotations given in FPTN. Authors affirm that their approach could be compliant with a MDA/MDE guidelines; however, they do not provide any MOF 2.0 compliant meta-model capturing the FPTN and CFT concepts. Thus, no ready for trial tool-supported model-driven method is available.

In [4], authors also propose a tool-supported model-driven method for the safety assessment of component-based architectures. Their approach makes it possible to extend Subsystems in Matlab/Simulink with complete component specifications and realizations and to analyze the extended Simulink models.

In [3], authors present a well conceived tool-supported model-driven method which generates FTA and FMECA tables from UML-based models. This approach supports forward as well as backward analysis. The tool-chain, however, consists of an integration of non-and-commercial tools and thus it may constitute an economical burden for industries.

In [9], authors provide a comparative study concerning model-driven and architecture-based safety evaluation methods. In particular, authors compare modelling approaches (such as FPTN, FPTC, HiP-HOPS) suitable for supporting dependability, more specifically safety, analysis. The main criteria of the comparison are: modelling support, process support and tool support. From this comparative works it emerges that: from a process support point of view almost all approaches considered present significant limitations; from a tool support point of view it emerges the potential of FPTC with respect to other approaches in enabling the possibility of generating FTA as well as FMEA tables for models that contain cycles; finally, from a modelling support point of view, a careful trade-off between expressivity and usability is mandatory. Stateless and syntactically balanced modelling approaches may lead to less modelling errors.

CHESS-FLA differs from the previous works in many ways: 1) it builds on a failure logic method that unifies forward and backward safety analysis techniques (e.g. FMECA and FTA) and goes beyond the limitations of FPTN and HiP-HOPS, discussed in Section II.A; 2) it builds on popular OMG standards; it provides a well integrated and open-source tool support for a promising FL analysis method; finally, it provides a detailed methodological support.

VI. Conclusion and future work

In this paper, we have proposed CHESS-FLA, a tool-supported model-driven dependability analysis method for component-based architectures. This method supports designers as well as dependability experts during the design of dependable component-based software architectures. Thanks to CHESS-FLA, a careful selection and composition of components is made possible. Designers have at disposal modelling facilities to focus on functional properties (functional view), dependability experts have at disposal modelling facilities to focus on dependability properties (extra-functional view) and to carry out dependability analysis (analysis view). Finally, jointly designers and dependability experts can interpret the analysis results and plan, if needed, the restructuring of the architecture, making it evolve into a more dependable one. To illustrate the usage and assess its usefulness, we have applied CHESS-FLA onto a simplified component-based packet transmission system.

CHESS-FLA represents an achievement that open up future research directions aimed at extending as well as improving it. Besides, CHESS-FLA, the CHESS-toolset supports other dependability analysis techniques (e.g. FTA and FMEA). Industrial case-studies have been specified and analyzed by using the different approaches offered by the tool-set and CHESS-FLA, as documented in the project

deliverables, has been judged helpful in supporting the design phase. In the short-term future, we plan to exploit these case-studies to provide comparative studies to show, when possible, equivalence in terms of results. Comparative studies are necessary to provide proofs/evidence to convince certification authorities and standardization bodies to accept automated and non-traditional analysis as a basis for certification.

In a medium-term period, we plan to compare the meta-model of CHESS-ML with the recently introduced meta-model for failure logic modelling approaches [12]. The aim of the comparison is the identification of possible missing language constructs necessary to cover all the relevant failure modelling approaches currently available. Relevance for us will mean tool-supported approaches easily integrable in the framework of our model-driven approach.

We also aim at making the syntactical rules more user-friendly, as well as, providing constraints to avoid contradictions. More specifically, similarly to what has been done in [18] we plan to extend FI^{4}FA syntactical rules to give users the possibility to reduce time effort, by having at disposal means to express the concept of any port to be used whenever it is not relevant on which input port a failure occurs. Concerning restrictions, we plan to prevent users from specifying contradicting behaviours whenever it is known that specific failure types cannot coexist (e.g. omission and commission as well as early and late).

ACKNOWLEDGMENT

This work has been partially supported by the European Project ARTEMIS-JU100022 CHESS [2]. The authors thank Stefano Puri for fruitful discussions on CHESS-FLA implementation details.

REFERENCES

[1] Acceleo - transforming models into code, http://www.eclipse.org/acceleo.

[2] ARTEMIS-JU-100022 CHESS- Composition with guarantees for High-integrity Embedded Software components aSsembly. https//www.artemis-ju.eu/chess.

[3] M. A. de Miguel, J. F. Briones, J. P. Silva, A. Alonso. Integration of safety analysis in model-driven software development. IET Software, pp. 260-280, 2008.

[4] D. Domis and M. Trapp. Integrating Safety Analyses and Component-Based Design. In proceedings of the 27th International Conference on Computer Safety, Reliability and Security, Newcastle on Tyne, UK, 22-25 September, 2008.

[5] P. Fenelon and J. A. McDermid. An integrated toolset for software safety analysis. Journal of Systems and Software, vol. 21, no. 3, pp. 279-290, 1993.

[6] B. Gallina, N. Guelfi. Reusing Transaction Models for Dependable Cloud Computing. Software Reuse in the Emerging Cloud Computing Era, pp 248-277, IGI Global, Editor(s): Hongji Yang and Xiaodong Liu, April 2012.

[7] B. Gallina and S. Punnekkat. FI^{4}FA: A Formalism for Incompletion, Inconsistency, Interference and Impermanence Failures Analysis. In proceedings of the IEEE International workshop on Distributed Architecture modeling for Novel Component based Embedded systems, Oulu, Finland, 2 September, 2011.

[8] B. Gallina, A. Dimov, S. Punnekkat. Fuzzy-enabled Failure Behaviour Analysis for Dependability Assessment of Networked Systems. IEEE International Workshop on Measurement and Networking, IEEE Computer Society, Anacapri, Italy, August, 2011.

[9] L. Grunske, J. Han. A comparative study into architecture-based safety evaluation methodologies using AADL's Error Annex and failure propagation models. 11th IEEE High Assurance Systems Engineering Symposium (HASE), pp.283-292, 2008.

[10] L. Grunske, B. Kaiser. Automatic generation of analyzable failure propagation models from component-level failure annotations. In Proceedings of the 5th International Conference on Quality Software, pp. 117- 123, 19-20 September, 2005.

[11] M. A. Javed and F. UL Muram. A framework for the analysis of failure behaviors in component-based model-driven development of dependable systems. Master thesis, Mälardalen University, School of Innovation, Design and Engineering, November, 2011.

[12] O. Lisagor. Failure Logic Modelling: a Pragmatic Approach. PhD thesis, University of York, 2010.

[13] O. Lisagor, J.A. McDermid, D.J. Pumfrey. Towards a Practicable Process for Automated Safety Analysis. In proceedings of the 24th International System Safety Conference, System Safety Society, pp. 596–607, Albuquerque, NM USA, 2006.

[14] L. Montecchi, P. Lollini, A. Bondavalli. Dependability Concerns in Model-Driven Engineering. In proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, 2011.

[15] R. F. Paige, L. M. Rose, X. Ge, D. S. Kolovos, and P. J. Brooke. FPTC: automated safety analysis for domain-specific languages. In Models in Software Engineering, M. R. Chaudron (Ed.). Lecture Notes In Computer Science, Vol. 5421. Springer-Verlag, Berlin, Heidelberg, pp. 229-242, 2009.

[16] Y. Papadopoulos. Hierarchically Performed Hazard Origin and Propagation Studies. In proceedings of the 18th International Conference on Computer Safety, Reliability, and Security, LNCS-1698, pp. 139-152, Springer-Verlag, 1999.

[17] M. Wallace. Modular architectural representation and analysis of fault propagation and transformation. Electronic Notes in Theoretical Computer Science, volume 141 n.3, pp.53-71, December, 2005.

[18] I. Wolforth, M. Walker, Y. Papadopoulos. A Language for Failure Patterns and Application in Safety Analysis. In Proceedings of the 3rd IEEE International Conference on Dependability of Computer Systems, pp.47-54, Szklarska Poreba, Poland, 26-28 June, 2008.

[19] W. Wu and T. Kelly. Failure modelling in software architecture design for safety. In proceedings of the ACM ICSE Workshop on Architecting Dependable Systems (WADS), St Louis, Missouri, USA, May, 2005.

[20] F. Ye and T. Kelly. Component Failure Mitigation According to Failure Type. In Proceedings of the 28th IEEE Annual International Computer Software and Applications Conference, 2004.