Towards a MDE Transformation Workflow for Dependability Analysis

Leonardo Montecchi, Paolo Lollini, Andrea Bondavalli
Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze
Firenze, Italy
{lmontecchi, lollini, bondavalli}@unifi.it

***Abstract*—In the last ten years, Model Driven Engineering (MDE) approaches have been extensively used for the analysis of extra-functional properties of complex systems, like safety, dependability, security, predictability, quality of service. To this purpose, engineering languages (like UML and AADL) have been extended with additional features to model the required non-functional attributes, and transformations have been used to automatically generate the analysis models to be solved by appropriate analysis tools. In most of the available works, however, the transformations are not integrated into a more general development process, aimed to support both domain-specific design analysis and verification of extra-functional properties. In this paper we explore this research direction presenting a transformation workflow for dependability analysis that is part of an industrial-quality infrastructure for the specification, analysis and verification of extra-functional properties, currently under development within the ARTEMIS-JU CHESS project. Specifically, the paper provides the following major contributions: i) definition of the required transformation steps to automatically assess the system dependability properties starting from the CHESS Modeling Language; ii) definition of a new Intermediate Dependability Model (IDM) acting as a bridge between the CHESS Modeling Language and the low-level analysis models; iii) definition of transformations from the CHESS Modeling Language to IDM models.**

Keywords-model driven engineering, UML, transformation workflow, intermediate model, dependability analysis, CHESS

I. INTRODUCTION

The systematic use of models as primary artefacts throughout the engineering lifecycle is the basic principle behind Model Driven Engineering (MDE) [1]. In the last decade, MDE approaches have been extensively applied to support the analysis of the so-called extra-functional system properties, e.g., safety, dependability, security, predictability, quality of service. To allow a non-ambiguous specification and design, constraints are added to engineering languages like UML and AADL, which are then used as input for subsequent development steps like code generation, formal verification, testing and assessment. Language extensions, like the UML profile for QoS and Fault Tolerance [2], were introduced and used to capture the required extra-functional concerns. System designers use the language extensions to identify the component types and assign local dependability parameters to hardware and software artifacts in the engineering model. Then, automated tools (mainly based on pattern matching and model transformation) are used to assemble the relevant sub-models on the basis of the system structure, and to invoke the appropriate algorithms that solve the system-level model.

In the literature there are several works adopting model-driven engineering approaches for dependability analysis, but most of them are not designed to be part of a more general development process that supports both domain-specific design, analysis and verification of extra-functional properties. In this paper we present a MDE transformation workflow for dependability analysis, which is part of the modeling framework currently under development within the ARTEMIS-JU CHESS project [3]. The philosophy behind CHESS is to adopt MDE approaches to achieve higher degrees of abstraction and automation in support of the whole development process, thus including validation, and combine them with component based development techniques for architectural design purposes, to increase the reusability of software assets and to ease the maintenance. In this context, the paper offers the following main contributions:

- Definition of the overall transformation workflow, from the high-level CHESS Modeling Language (CHESS ML) to the low-level dependability analysis models, with back-annotation of the obtained results.
- Definition of an Intermediate Dependability Model (IDM), which acts as a bridge between CHESS ML and the low-level analysis model and allows the representation of advanced dependability features.
- Definition of the transformations from CHESS ML to IDM, which allow to extract the system characteristics required for the dependability analysis and organize them in a proper structure.

The rest of the paper is organized as follows. Related works concerning model-driven engineering approaches for dependability analysis are discussed in Section II. Section III introduces the CHESS framework, and the related modeling language. Section IV describes the workflow for dependability analysis adopted within the project, while the core of this approach, the Intermediate Dependability Model, is detailed in Section V. In Section VI a set of model transformations to derive an IDM representation of CHESS models is defined. Finally, conclusions and next steps are drawn in Section VII.

II. RELATED WORK

Several works in the literature adopt a model-driven engineering approach to perform dependability analysis. Following MDE principles, the model of the system in some specific analysis language (e.g., Stochastic Petri Nets) is automatically derived from a higher-level description of the system developed using more abstract languages like UML.

The idea of translating UML models to dependability models was elaborated and refined in several papers. In [4] a fault tree of the system is derived processing a set of UML diagrams. In [5] UML models are enriched with probability values, and the system's failure is evaluated using Bayesian rules. A framework for the evaluation of distributed systems has been defined in [6], where the analysis model is derived from an overall model composed of an UML model and a network topology description.

In [7] structural UML diagrams form the basis of a transformation to Timed Petri Net dependability models. Performability and dependability models are instead constructed on the basis of behavioral UML diagrams in [8]. Here the analysis model is generated from guarded statecharts, i.e., statechart diagrams where transitions are labelled with guards. Event-based systems are covered in [9], where a transformation from statechart diagrams to Stochastic Reward Nets is presented. In a hierarchical modeling approach, this behavioral-level transformation can be effectively used to construct the sub-models of redundancy managers, whose behavior determines replica management and service restoration (recovery) [10]. Tools that implement transformation approaches have been also developed; as an example, the OpenSESAME tool [11] uses high-level (graphical) diagrams to express dependencies and transforms them to Stochastic Petri Nets.

Most of the works adopting MDE principles for dependability analysis define a direct transformation from the high-level architectural model to the analysis model. The resulting transformation rules are usually characterized by low flexibility (i.e., they are hard to adapt to changes in the target languages) and low reusability (i.e., they are hard to adapt to different languages). The HIDE [12] and PRIDE [13] projects addressed these issues using an intermediate dependability model acting as a bridge between the high-level modeling language and the dependability analysis formalism.

The intermediate model introduces an additional abstraction layer, through a representation that is independent from both the engineering modeling language and the analysis formalism. Although the introduction of an additional transformation step might seem to add unnecessary complexity, the definition of the two transformations will typically require less effort than the definition of a single, monolithic, one. Moreover, the adoption of an intermediate model generates more flexible transformations: should one of the two languages (i.e., the high-level language or the analysis formalism) change, only the transformation rules for that language would be affected, leaving the rules for the other side unchanged. In addition, if we consider n engineering languages and m analysis formalisms, $n \times m$ possible transformations between them exist; conversely, using an intermediate model, only $n + m$ transformation rules are required to cover all the possible combinations.

Despite several approaches that use model transformations for dependability analysis exist, the process that is adopted is often very language-specific or bound to the particular application domain under analysis. What is currently missing in the development of critical and embedded systems is a comprehensive framework that integrates the support for automated dependability analysis together with more generic design, development and assessment facilities. In the development of critical and embedded systems many non-functional constraints are imposed on the system's structure and behaviour, spanning different points of view. Specific analysis methods are then used to assess the feasibility of a certain system design with respect to such requirements; moreover, non-functional requirements may generate conflicts and therefore analyses are needed also to select the proper tradeoffs.

In this perspective, dependability attributes are a subset of the non-functional properties that must be specified, analysed and verified during the system development process. Dependability itself concerns with different aspects of the system (e.g., reliability, availability, safety) that may be addressed by different analysis techniques. Managing the complexity that arises from such heterogeneity of properties, techniques, and tools requires an integrated modeling framework that ensures interoperability and consistency between different aspects of the development process.

Some work in this direction was carried out within the HIDE project, in which different analysis models could be derived from the UML model of the system, addressing both quantitative and qualitative dependability analysis. Other works focused on the definition of UML extensions to support different dependability analysis. The definition of the DAM profile [14], an UML profile for dependability analysis based on MARTE [15], is one of the most interesting works aiming to create a single dependability modeling language that accommodates dependability attributes required by different analysis methods. The attributes included in DAM are defined merging in a single profile all the attributes used in different works surveyed from the literature, but this strategy has produced a language where the user is allowed to introduce inconsistencies and ambiguities in the model, not having any guidance or constraints to help him.

Within the ongoing CHESS project, the inconsistencies are solved at the conceptual level, through an iterative process in which common or related concepts needed for different types of analysis are merged in a single conceptual element. The resulting conceptual model is not yet a profile, but it contains all the concepts that need to be somehow (syntactically) represented at engineering language level.

III. THE CHESS FRAMEWORK

The CHESS project aims at developing, applying and assessing an industrial-quality Model-Driven Engineering infrastructure for the specification, analysis and verification of extra-functional properties (predictability, dependability and security) in component-based systems modeling. The methodology defined in CHESS should be suitable for different application domains including (but not limited to) automotive, railway, and space.

The CHESS philosophy refers to a particular MDE initiative, the Model-Driven Architecture (MDA) defined by OMG [16]. In the workflow promoted by MDA the system designer creates a Platform Independent Model (PIM), which is independent from the execution platform that will actually implement the system. From the PIM model, enriched with

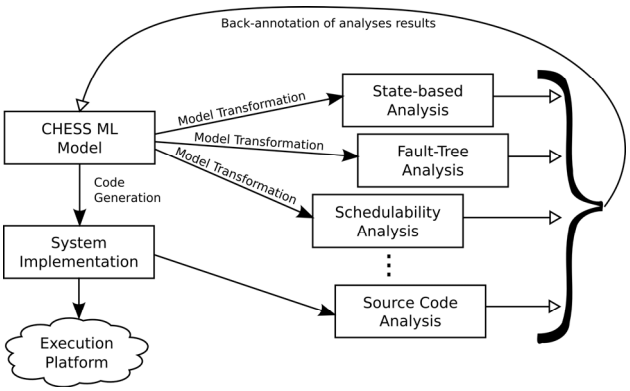

Figure 1. The CHESS workflow for system development and analysis

deployment information, a Platform Specific Model (PSM) is then generated by automated transformations. From PSM, code generation may be triggered to obtain an implementation of the system.

The development process is supported by different kinds of analyses (e.g., dependability, schedulability), which allow assessing the feasibility of the system's design with respect to different aspects. In accordance with MDE principles, the analysis models are automatically derived from the high-level model that describes the system's architecture. Following such approach, the user has not to specify the details of the analysis model of the system, which can be a tedious and very error-prone process. Moreover, in this way the construction of the analysis model takes advantage of the knowledge of specific experts, which otherwise may not be available to the system designer. Analysis' results are then used to enrich the initial CHESS ML model.

The CHESS framework supports and promotes an iterative development process, in which the system's model is constantly updated and refined based on the results obtained by different analysis techniques (see Fig. 1). Analysis results are annotated back in the system model, and can be used as input for subsequent analyses. Code generation techniques can be used to automatically generate a system implementation for a given execution platform. The source code, possibly including legacy code, can be analyzed as well using code analysis techniques (e.g., call-graph analysis).

A. CHESS Modeling Language

Practical support to the CHESS methodology is provided by the CHESS Modeling Language (CHESS ML), a high-level modeling language that is built from subsets of standard languages: UML, SysML and MARTE. The definition of the language is performed as an iterative process in which it is constantly refined and harmonized, addressing inconsistencies and clashes between different domains of interest.

Concerning dependability analysis, the CHESS methodology supports several analysis methods representing the system at different abstraction levels and having different objectives. Among them:

- Failure Modes, Effects and Criticality Analysis (FMECA), which aims to identify each potential failure within a system or manufacturing process and uses severity classifications to show the potential hazards associated with these failures;
- Fault Propagation and Transformation Calculus (FPTC), which analyses the failure behavior of a component based on the failure modes of its sub-components;
- State-based analysis, which allows evaluating dependability and performance attributes of the system, taking into account for complex relationships between components.

Each method requires the system model to be extended with a set of specific dependability attributes, possibly shared between different methods. For example, the concept of "failure" is shared between all the analysis methods mentioned above. One of the challenges in applying this approach is to ensure that the UML designer has a correct view of the meaning of the input attributes and of the role they play within the individual analysis methods. An incorrect understanding could actually lead to a wrong interpretation of the analysis' results, possibly leading to wrong design decisions.

It is therefore paramount to avoid inconsistencies and duplication of attributes in the high-level model; for this purpose the concepts needed by each analysis method are merged into a conceptual model, i.e., a single semantic space, where overlapping and inconsistencies can be addressed before the definition of the actual language extensions. Possible inconsistencies between attributes, due to the heterogeneity of analysis techniques, are resolved at the conceptual level, thus reducing the number of elements in the profile. Further details on the CHESS approach in supporting dependability concerns in MDE can be found in [17].

Such concepts are then instantiated into the CHESS Dependability Profile, which allows to enrich a CHESS ML model with information related to dependability. The CHESS Dependability Profile, as well as the features addressing other concerns (e.g., schedulability), will contribute to the final specification of the CHESS Modeling Language.

To provide support for the overall CHESS methodology, a set of ad-hoc plugins for the Eclipse platform are being developed within the project, including a diagram editor based on a customized version of MDT/Papyrus [18], a promising EMF graphical model editor focused on UML and derived languages.

B. CHESS ML elements supporting state-based analysis

Among the different techniques for dependability analysis supported by CHESS ML, this paper focuses on state-based methods. In this kind of analysis the system is modeled using a state-based formalism, e.g., Continuous Time Markov Chains (CTMCs) or Stochastic Petri Nets (SPNs). These kinds of models provide a representation of the system's state and its possible changes with respect to time, allowing to model more complex interaction between components than using simpler combinatorial formalisms like fault-trees. Dependability-related measures are evaluated assessing the probability of the system of being in a certain state; performance-oriented measures can be evaluated enriching the model with costs and rewards.

To perform state-based dependability analysis, the archi-

TABLE I. MAIN CHESS ML ELEMENTS SUPPORTING STATE-BASED DEPENDABILITY ANALYSIS.

StatefulHardware	StatelessHardware	StatefulSoftware	StatelessSoftware	Propagation	FailureMode	DerivedFailure	FaultTolerant
faultOccurrence	faultOccurrence	faultOccurrence	faultOccurrence	prob	name	name	redundancyScheme
errorLatency	probPermFault	errorLatency	transDuration	propDelay	description	description	schemeAttributes
probPermFault	repairDelay	repairDelay			relProp	relProp	
repairDelay	transDuration	transDuration				propLogic	
transDuration							

tectural model is enriched with the required information, using a subset of the attributes and stereotypes defined in the CHESS Dependability Profile. The main elements used in the (automated) construction of the state-based dependability model are summarized in Table I, and their description is provided in the following.

1) StatefulHardware", "StatelessHardware", "StatefulSoftware", and "StatelessSoftware".

The above stereotypes are used to classify system components according to their behaviour with respect to faults, errors, failures, and repair. Components can be marked as hardware or software, and as stateful or stateless; based on the combination of these two dimensions the four above classes are derived. They share a "faultOccurrence" attribute, which specifies the rate of fault occurrence, which is assumed to follow an exponential distribution.

Stateful components have an internal state, and therefore a fault does not immediately lead to a failure, but it may generate latent errors that require additional time to reach the service interface. Such time delay (again assumed to be exponentially distributed) is specified by the "errorLatency" attribute. Stateless components do not have an internal state and a fault occurrence immediately causes the failure of the component. Hardware components may be affected by both permanent and transient faults; the "probPermFault" attribute specifies the probability that a fault that develops in the component is a permanent fault. Conversely, for software components it is reasonable to assume that they are only subject to transient faults during their operation, and that permanent faults have been removed by thorough testing and debugging activities in the development process [19]. For each component, the duration of a transient fault follows an exponential distribution, whose mean is specified through the "transDuration" attribute.

A failed component is repaired after a given amount of time has elapsed. This amount of time is again assumed to follow an exponential distribution, whose mean is specified by the "repairDelay" attribute. StatelessSoftware components are immediately repaired after a failure has occurred: they are not affected by "physical" degradation after failure, neither they have an internal state that may become erroneous; for this reason, they are immediately functioning again after a failure has occurred.

2) "FailureMode"

The CHESS Dependability Profile allows to associate multiple failure modes with system components. By default, a single failure mode is associated with each component, but the user may extend the basic model defining additional failure modes. The "FailureMode" element represents a failure mode of an atomic component, i.e., a component that does not have a refinement in the model, or for which such refinement is not considered. Each failure mode has a "name" and a "description", which are used to identify and describe it. An additional attribute, "relProp", specifies the relative probability of one failure mode with respect to the others that are associated with the same component.

3) "DerivedFailure"

The failure of composed components can be caused by the failure of their subcomponents. By default, a composed component is considered failed if at least one of its subcomponents is failed. The user may however extend the basic model defining additional failure modes for composed components as well. A "DerivedFailure" element is defined as a specialization of "FailureMode", and therefore it inherits all its attributes. Additionally, a "DerivedFailure" has a "propLogic" attribute, which specifies the logical condition on the failure of subcomponents that generates the failure of the higher-level component (i.e., the derived failure).

4) "Propagation"

In the construction of the dependability model, most propagation paths are derived from relations that are described in the functional model of the system (e.g., deployment relations, client/server relations, etc.). The "Propagation" stereotype allows to enrich such relations with additional information related to dependability analysis. More in detail, this stereotype allows to specify the probability that error propagation takes place through the relation ("prob" attribute), and the time delay after which propagation takes place ("propDelay" attribute). If such attributes are not specified we assume that error propagation always takes place (i.e., with probability equal to one), and that it takes place instantaneously (i.e., with zero propagation delay).

5) "FaultTolerant"

This stereotype is used to identify system components that are implemented by fault tolerant structures. Two attributes allow to specify the details of the redundancy scheme that is implemented by the fault tolerant structure. The "redundancyScheme" attribute allows to specify the kind of fault tolerant structure (e.g., TMR, 2oo2, etc); the "schemeAttributes" attribute allows to specify the additional parameters that characterize the selected redundancy structure (e.g., the coverage of the voter in a TMR structure).

IV. STATE-BASED DEPENDABILITY ANALYSIS WORKFLOW

The workflow for state-based dependability analysis is inspired by the approach formalized in [12] and successively refined in [13]. The distinctive feature of these works with respect to the literature on model transformations for dependability analysis is the introduction of an intermediate dependability model, an intermediate representation of the system that abstracts both from the high-level engineering description and from the low-level implementation in the selected analysis formalism.

Following this philosophy, the transformation from

CHESS ML to a state-based dependability analysis model is defined as a multi-step transformation process:

1. The first step has the fundamental task of extracting the required dependability information from the mass of information available in the CHESS ML model. The model elements and relations in CHESS ML are mapped to elements and relations of the Intermediate Dependability Model, obtaining a model of the system which has only the information that is required for state-based dependability analysis.
2. The second step translates the intermediate representation to a general, high-level, representation of a SPN model. The candidate language to express this general Petri net model is the Petri Net Markup Language (PNML) [20], a proposal of a Petri net interchange format based on XML that is under development as an ISO standard. The benefit of using PNML is twofold: on one hand it is expected to provide better tool support, if analysis tools are going to support it; on the other hand it still allows greater flexibility, providing a general Petri net representation that is not bound to any specific analysis tool.
3. The third step generates the input for the selected analysis tool based on the PNML description. The tool can then be executed to perform the desired evaluations. If the selected tool directly supports PNML then this step can be skipped.
4. The last step performs the back-annotation of evaluation results into the original CHESS ML model. Back-annotation allows for incremental model construction: this new information included in the CHESS ML model can be used as input for subsequent analyses.

The core of this approach is the Intermediate Dependability Model (IDM), which serves as a fixed reference point between all the variable elements: the analysis formalism, the analysis tool, and even the high-level modeling language.

V. THE INTERMEDIATE DEPENDABILITY MODEL (IDM)

In this section we provide a high-level description of our new Intermediate Dependability Model (IDM), focusing on its capabilities to represent the dependability properties of the system that are required for state-based dependability analysis. With respect to previous approaches based on intermediate models, our new IDM provides additional modeling features and an improved modeling power. More in detail, the new features that have been introduced allow to:

- define multiple failure modes for system's components;
- characterize failures based on their domain, detectability, consistency and consequences;
- model internal error propagation and possible error compensation;
- model the details of faults, errors, and failures chain inside components;
- model preventive and corrective maintenance activities;
- model error detection mechanisms;
- relate maintenance activities to the results of error detection activities;
- describe the details of the measures of interest that should be evaluated on the system.

IDM also includes the definition of some of the most used probability distributions in dependability analysis, which can be associated with all the timed quantities within the model. An in-depth description of the IDM metamodel is provided as a technical report in [21], while a simple IDM model of a fire detection system is provided in [17].

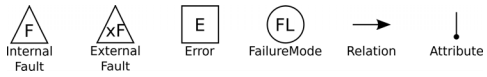

Figure 2. The graphical notation used for the main elements of the IDM.

The IDM representation is composed of nodes and relations, and it can be conveniently expressed using a graphical notation. As shown in Fig. 2, different nodes are distinguished by their shape: faults are represented by triangles, errors by squares and failure modes by circles. This distinction permits to easily identify the elements involved in propagation paths. Relations between two elements of the model are represented by an arrow following the direction of the relation, while attributes are represented by short lines ending with a dot.

A. *Modeling single components*

In the IDM representation the system's components are modeled by "Component" elements, which may have a certain number of "Fault", "Error" and "FailureMode" elements associated with them. An example of IDM model of two atomic components A and B is shown in Fig. 3, which includes a propagation relation as well. Faults that develop within components are modeled by "InternalFaults" elements; the time delay after which a fault occurs is given by the "Occurrence" attribute, which specifies the probability distribution of the occurrence delay. Internal faults may be permanent or transient with a given probability, specified by the "PermanentProbability" attribute.

When a fault is activated, after a certain delay it generates an error within the component. These propagation paths are specified by "FaultsGenerateErrors" relations, which connect "Fault" elements with "Error" elements. Additional attributes are used to specify the delay after which the propagation takes place ("ActivationDelay" attribute), the probability that propagation actually occurs ("PropagationProbability" attribute), and the weight of this propagation path with respect to the others that originate from the same set of faults ("Weight" attribute).

After a certain amount of time errors may disappear or get compensated for effect of computation. The optional attribute "VanishingTime" specifies the probability distribution of the amount of time after which an error is compensated and it disappears from component's state. If an error does not disappear, it may further propagate, possibly reaching the component's interface, thus causing a failure of the component. A component may be affected by different failure modes, which are modeled by "FailureMode" elements. Failure modes are characterized according to four dimensions: domain, detectability, consequences and consistency. Propagation from errors to failure modes is modeled by "Er-

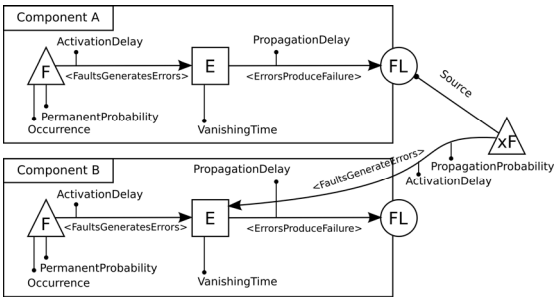

Figure 3. IDM model of error propagation from Component A to Component B.

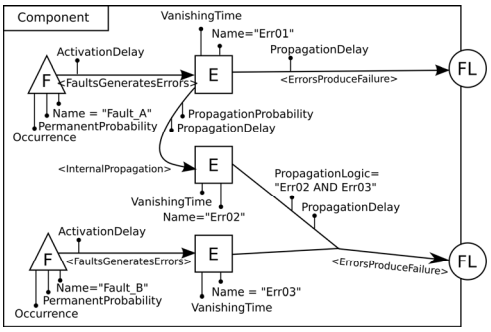

Figure 4. IDM model of complex propagation path within a single component, also involving internal error propagation.

rorsProduceFailures" elements, which connect "Error" elements with "FailureMode" elements within the same component. Similarly to "FaultsGenerateErrors" relations, "ErrorsProduceFailures" relations are characterized by a delay after which propagation takes place ("PropagationDelay" attribute), a propagation probability ("PropagationProbability" attribute), and a weight ("Weight" attribute).

B. Modeling error propagation

Representing interactions of components and error propagation between them is paramount in dependability analysis. When a component uses the service provided by another component, it may be affected by error propagation in case of service failure. In the IDM representation, error propagation between different components (external propagation) is modeled through the use of external faults: a failure of a system's component is perceived as a fault by the components that rely on it.

An example of IDM model that includes a propagation relation is provided in Fig. 3. When modeling error propagation in the IDM representation, the "FailureMode" that generates the propagation is connected to an "ExternalFault" element, which is then associated with the component that is affected by the propagation. Similarly as for internal faults, an external fault may propagate and generate an error in the target component; this kind of propagation is specified by "FaultsGenerateErrors" relations. In this case the "PropagationProbability" and "ActivationDelay" attributes correspond, respectively, to the probability that the propagation between the two components occurs, and the delay which is needed for a failure of the server component to generate an error in the client component.

C. Modeling internal propagation paths

The IDM language allows to define complex propagation chains within components, involving multiple faults, errors and failure modes. Multiple instances of "Fault" (either internal or external), "Error" and "FailureMode" elements may be associated with "Component" elements. An example of IDM model comprising complex internal propagation paths is provided in Fig. 4.

As mentioned above, "FaultsGenerateErrors" and "ErrorsProuceFailures" relations specify how "Fault", "Error", and "FailureMode" elements are interconnected. Logical conditions may also be attached to such relations, to further specify the conditions that cause the propagation to take place ("PropagationLogic" attribute).

Moreover, our intermediate model allows error propagation to take place within the component boundaries (e.g., within a CPU as an effect of computation). More in detail, "InternalPropagation" relations may be used to connect "Error" elements to other "Error" elements within the same component. Similarly to the other propagation relations, "InternalPropagation" relations are characterized by a "PropagationDelay", a "PropagationProbability", a "PropagationLogic" and, possibly, a "Weight".

D. Modeling error detection and maintenance activities

A specific package of the IDM, the Maintenance & Monitoring package, allows to provide an intermediate representation of monitoring activities and maintenance activities (both preventive and corrective). We use the term "monitoring" here in its most general sense, meaning any activity on the system that provides information on the state of the system or on one of its components. An example of IDM model including maintenance and monitoring activities can be found in [21].

Elements of type "DetectionActivity" represent error detection activities that are performed on system's components. The attribute "DetectableErrors" specifies the error(s) that the activity tries to detect, which may also belong to different components. The "Duration" attribute specifies the probability distribution of the time needed to perform the activity, while the "When" attribute specifies when the activity should be executed, i.e., the schedule of the activity. The schedule of the activity is specified by particular kinds of expressions ("ScheduleExpression") that provide both a time constraint and a logical expression on the state of the system. "DetectionActivity" elements are also characterized by a "Coverage" attribute and a "FalseAlarmRatio" attribute, which describe the quality of the detection mechanism that is modeled by the activity.

Concerning maintenance, "RepairActivity" and "ReplaceActivity" elements are used to model repair and replace, respectively. They share a "Target" attribute, which specifies the "Component" elements that are the objective of the maintenance activity (i.e., the components that are repaired or replaced). "ReplaceActivity" elements have also a "Replacement" attribute associated with them, which can be used to provide further details on the component that is used as replacement (e.g., if it has different properties with respect to the replaced component); such situation may occur for

example when a component is upgraded to a newer version. The attribute "SuccessProbability" specifies the probability that the maintenance activity completes successfully; if the activity completes successfully, the target element is repair/replaced as intended, otherwise it is left in a faulty state.

Maintenance and monitoring activities may be executed by other components within the system. "ComponentExecuteActivity" relations are used to connect activities with the "Component" that is in charge of executing them. A failed component will not perform any monitoring and maintenance activity.

E. Modeling the hierarchical structure of the system

In order to analyze the whole system it is necessary to specify its hierarchical structure, i.e., how components are logically or physically grouped to form higher-level components. As an example, Fig. 5 shows the IDM model of a composed component "A&B", which is considered failed if one of its subcomponents A or B are failed. When a component is composed of subcomponents (either logical or physical) and subcomponents are included in the dependability model, any failure of them should be treated as an external fault of the higher-level component. Therefore, in the IDM representation composed components are modeled through external faults: the composed component itself is represented by a simple "Component" model element, in the same way as for atomic components. To model the dependency from its subcomponents, "ExternalFault" elements are associated with it, and connected to the "FailureMode" of its subcomponents through the "Source" attribute.

This kind of propagation is then treated in the same way as the other propagation relations (e.g., propagation between two interacting components).

F. Modeling the measures of interest

One of the features included in our new Intermediate Dependability Model is the possibility to define different measures of interest to be evaluated on the system's model. Fig. 5 shows an IDM model where two measures of interest are specified on the composed component previously detailed. At this stage the IDM supports the definition of a set of most common dependability measures, related to availability, reliability, and safety; thanks to the flexible architecture of the IDM metamodel new measures could be included in the future as well.

Reliability, availability, and safety measures are specified through the "Reliability", "Availability", and "Safety" model elements respectively. All these three elements are specialization of an abstract "DependabilityMeasure" element, from which they inherit the "Target", "Evaluations", "RequiredMin" and "RequiredMax" attributes. The "Target" attribute allows to select the subset of the failure modes associated to a component that should be targeted by the analysis. The "Evaluations" attribute is used to specify the set of evaluations that should be performed for the measure. An evaluation can be of one of the following types: i) "SteadyState", which has no additional attributes; ii) "InstantOfTime", where the instant is specified by a "TimePoint" attribute; or iii) "IntervalOfTime", where the interval is specified by the

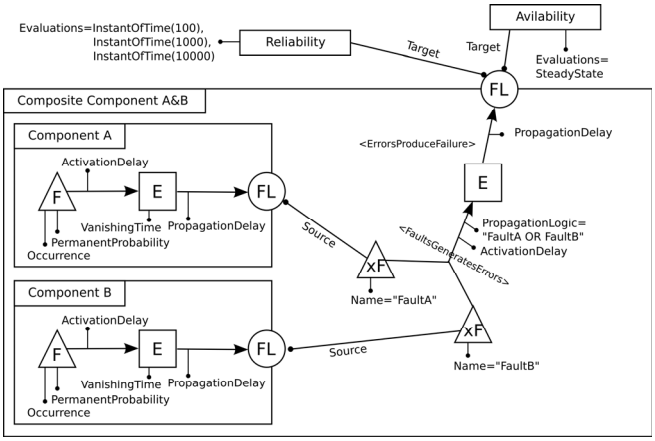

Figure 5. IDM model showing the intermediate representation of composed components and measures of interest.

"Begin" and "End" attributes. Finally, the "RequiredMin" and "RequiredMax" attributes can be used to specify constraints on the minimum or maximum value that is allowed for the measure; such constraints can be derived for example from system's dependability requirements.

VI. CHESS ML TO IDM TRANSFORMATIONS

In this section we provide a set of transformation rules to derive an IDM representation of the system, starting from a CHESS ML model. At this stage, CHESS ML and its profile for dependability analysis are still under definition, and for this reason a complete and formal definition of transformations cannot be provided. However, it is already possible to automatically map a subset of CHESS ML constructs to the intermediate dependability model.

For the sake of clarity, model elements belonging to the CHESS ML model are underlined, while model elements belonging to IDM are written in *italic*. To describe the attributes of model elements we use the "dot notation" commonly used in many programming languages.

A. Projection of atomic components

With respect to state-based analysis, an atomic component can be "StatefulHardware", "StatelessHardware", "StatefulSoftware", or "StatelessSoftware". For the purpose of state-based analysis we consider atomic components those that are marked with such stereotypes. The current version of the CHESS Dependability Profile assumes that quantities are only deterministic or exponentially distributed, and that components are subject to multiple failure modes, but all originating from a single fault and a single error. This means that, at this stage, not all the IDM capabilities can be actually exploited, since IDM allows to use more probability distributions and to define multiple faults and errors affecting the same component.

A "StatefulHardware" element is a hardware element that has an internal state. The projection of this element in the intermediate model generates a "Component" element, having a "Fault", an "Error" and a certain number of "FailureMode" elements associated with it. The number and the properties of "FailureMode" elements are based on the failure modes described in the CHESS ML model. The

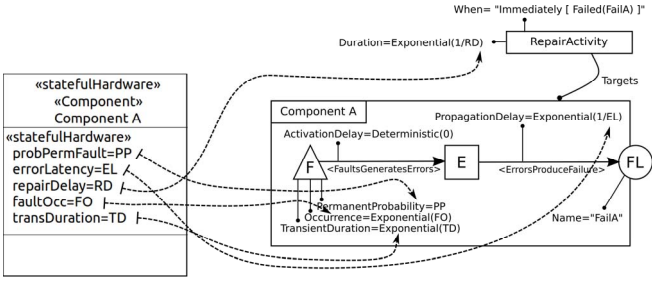

Figure 6. Projection of StatefulHardware elements in the IDM model.

attributes "faultOccurrence" and "probPermFault" are mapped to the "Occurrence" and "PermanentProbability" attributes of the Fault element, respectively. The "transientDuration" attribute is mapped to the "TransientDuration" attribute of the "Fault" element. The "errorLatency" attribute is mapped to the "PropagationDelay" attribute of an "ErrorsProduceFailures" relation. Finally, the repair delay is modeled using a "RepairActivity", having its "Duration" attribute equal to the value specified by the "repairDelay" attribute in the CHESS ML model. Since the component is repaired independently of what failure mode has occurred, the "Target" attribute of the "RepairActivity" references all the "FailureMode" elements of the involved component. The transformation is sketched in Fig. 6, for the case where the component is affected by only a single failure mode. The algorithmic description of this transformation is provided in Fig. 7.

The transformation rules involving "StatelessHardware" elements are very similar and are not fully described here for lack of space. The main difference is that the PropagationDelay attribute of the ErrorsProduceFailures relation(s) is always set to Deterministic(0), to model an instantaneous propagation. This is because stateless elements do not have an internal state, and thus the activation of a fault immediately leads to a failure of the component.

For what concerns software components, the transformation rules follow again a similar structure. However, since they are only subject to transient faults, the PermanentProbability attribute of the Fault element is always set to zero. Moreover, StatelessSoftware components do not have an internal state and therefore the PropagationDelay attribute of the ErrorsProduceFailures relation(s) is always set to Deterministic(0). For StatelessSoftware the repair delay is also instantaneous, since they are not subject to physical degradation like hardware components; for this reason the Duration attribute of the RepairActivity associated with the component is always set to Deterministic(0). These additional transformations are described in details in [21].

B. *Projection of composed components*

The transformation for composed components considers components for which a refinement of their internal structure exists in the CHESS ML model (e.g., a Composite Structure Diagram). The algorithm follows a depth-first pattern: when a refinement of a component is found, the transformation processes the subcomponents first, and then the composed structure. If subcomponents are composed components themselves, then lower-level components are processed first and so on, until atomic components are reached.

If the component has been marked as StatefulHardware, StatelessHardware, StatefulSoftware or StatelessSoftware then the transformation will not project anything new in the intermediate representation. In fact, in this case the component has been already processed when projecting atomic components. For each composed component in the CHESS ML model not marked with such stereotypes, a new "Component" element is created in the intermediate model. A certain number of "ExternalFault" elements are associated with such "Component", one for each "FailureMode" belonging to "Component" elements that represent the sub-components.

If not differently specified, composed components are considered failed if at least one of their subcomponents is failed. Therefore, if no "DerivedFailure" elements are associated with the composed component, only one "Error" and one "FailureMode" elements are associated with the composed component. All the "ExternalFault" elements are connected to the single "Error" element with a "FaultsGenerateErrors" relation, having as "PropagationLogic" all the faults connected by the OR operator. The "Error" is then connected to the "FailureMode" by an "ErrorsProduceFailures" relation.

Conversely, if different "DerivedFailure" elements have been defined for the composed component, multiple "Error"

- For each element sfh of type StatefulHardware in the CHESS ML model:
  - create an element *c* of type *Component* in the intermediate model;
  - create an element *fo* of type *Exponential*, having *fo.Rate* equal to the value of the faultOccurrence attribute of the sfh element;
  - create an element *td* of type *Exponential*, having *td.Rate* equal to the value of the transDuration attribute of the sfh element;
  - create an element *el* of type *Exponential*, having *el.Rate* equal to the inverse of the errorLatency attribute of the sfh element;
  - create an element *ft* of type *InternalFault*, having *fo* as *Occurrence*, *td* as *TransientDuration*, and probPermFault as *PermanentProbability*, and add it to *c.Faults*;
  - create an element *e* of type *Error* and add it to *c.Errors*;
  - create a relation fge of type *FaultsGenerateErrors*, having *fge.Source=ft*, *fge.Destination=e*, *fge.ActivationDelay=Deterministic(0)*, and *fge.PropagationLogic=ft*.
- If sfh does not have any FailureMode elements associated with it:
  - create a node *f* of type *FailureMode* and add it to *c.FailureModes*;
  - create a relation *epf* of type *ErrorsProduceFailures*, having *epf.Source=e*, *epf.Destination=f*, *epf.PropagationDelay=el*;
- otherwise:
  - for each FailureMode fm that is associated with sfh:
    - create an element *f* of type *FailureMode* and add it to *c.FailureModes*;
    - create an element *f* of type *FailureMode* and add it to *c.FailureModes*;
    - create a relation *epf* of type *ErrorsProduceFailures*, having *epf.Source=e*, *epf.Destination=f*, *epf.PropagationDelay=el*, *epf.PropagationLogic=e*;
    - assign to *epf.Weight* the value of the attribute fm.relProp
- finally:
  - create an element rd of type *Exponential*, having *rd.Rate* equal to the inverse of sfh.repairDelay;
  - create an element ra of type *RepairActivity*, having *ra.Target=c* and *ra.Duration=rd*;
  - set the attribute *ra.When* to the expression "*Immediately [Failed(f1) OR Failed(f2) OR ... OR Failed(fn)]*", where each *fk* is an element in *c.FailureModes*.

Figure 7. Transformation for StatefulHardware Components.

- For each relation conn of type UML::Connector that connects two components, client and server, where client is the component for which the port is "required" and server is the component for which the port is "provided", do the following;
- if the internal structure of server is detailed in the CHESS ML model and its subcomponents have a projection in the intermediate model:
  - If the port is delegated to the subcomponent subserver:
    - » stop and perform the transformation as if there was a direct connection between subserver and client;
  - otherwise:
    - » create an element *xft* of type *ExternalFault* for each *FailureMode fm* that is associated with server;
- otherwise:
  - create an element *xft* of type *ExternalFault* for each *FailureMode fm* that is associated with server;
- If the internal structure of client is detailed in the CHESS ML model and its subcomponents have a projection in the intermediate model:
  - If the port is delegated to the subcomponent subclient:
    - » stop and perform the transformation as if there was a direct connection between server and subclient;
  - otherwise:
    - » stop and perform the transformation as if there was a direct connection between server and all the subcomponents of client.
- otherwise:
  - create a relation *fge* of type *FaultsGenerateErrors*, with *fge.Source=xft* and *fge.PropagationLogic=xft*;
  - add to *fge.Destination* the *Error* element associated with the intermediate representation of client;
  - if the connector is stereotyped as Propagation, then set the attributes *fge.ActivationDelay* and *fge.PropagationProbability* to the values of the attributes conn.propDelay and conn.prob, respectively; otherwise set *fge.ActivationDelay=Deterministic(0)* and *fge.PropagationProbability=1*.

Figure 8. Transformation for Propagation Through Ports

and "FailureMode" elements are associated with the "Component", and each "Error" is connected to the corresponding "FailureMode" by an "ErrorsProduceFailures" relation. The set of "ExternalFaults" corresponding to subcomponents is connected to each "Error" by a "FaultsGenerateErrors" relation, where the "PropagationLogic" attribute is set based on the value of the "propLogic" attribute of the corresponding "DerivedFailure" in the CHESS ML model.

The algorithmic description of this transformation can be found in [21].

C. Projection of propagation relations

In this section we describe transformation rules that involve error propagation relations. Error propagation between components may take place essentially for two reasons:

- A component uses the service provided by another component. In this case a failure of the component that provides the service can generate an error in the "client" component. This kind of relation is modeled in CHESS ML through UML::Connectors relations.
- A software component is deployed on a hardware component. In this case a failure of the hardware component may generate errors in the software component, or directly cause its failure. This kind of relation is modeled in CHESS ML through MARTE::Allocate relations.

1) UML::Connector relations

In CHESS ML, components are connected through ports. An UML::Connector may connect two ports of two different components, or it may connect a port of one component to a port of one of its subcomponents. The former case describes a relation between two components at the same level and a possible error propagation path between the two; in the latter the functionality of the port is delegated to the subcomponent, which may then be affected by errors coming from that port. Ports may be input ports ("required"), output ports ("provided") or both. Error propagation originates from components that own output ports, and ends to the components connected to that port through their input ports. Whenever the port type is not specified, bidirectional error propagation is assumed.

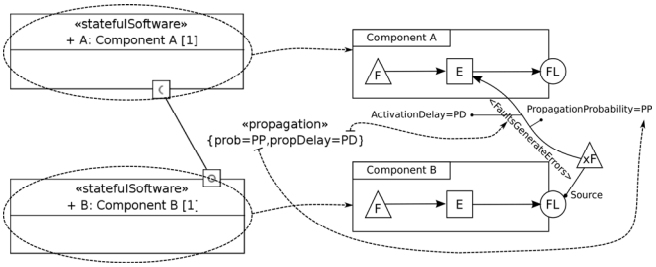

Figure 9. Projection of UML::Connector relations in the IDM representation

The projection of these relations creates an "ExternalFault" element for each "FailureMode" of the "server" component (i.e., the component for which the port is of type "provided"); the "ExternalFault" is then connected, through a "FaultsGenerateErrors" relation, to an "Error" element of the component that owns the input port. If the relation is stereotyped as "Propagation", then the attributes "PropagationDelay" and "PropagationProbability" are set to the values of the CHESS ML attributes "propDelay" and "prob", respectively; otherwise default values are assumed. A sketch of this transformation is given in Fig. 9, using a graphical representation. The algorithmic description of the transformation is provided in Fig. 8.

1) MARTE::Allocate relations

Allocation relations indicate a possible error propagation path, directed from the hardware component to the software that is using it. The projection of such relations is similar to the one for UML::Connector relations described above, considering that the software components "uses" the hardware component. The algorithmic description of this transformation is provided in [21].

VII. CONCLUDING REMARKS

The paper presented a MDE transformation workflow for the quantitative evaluation of dependability-related metrics. The approach is integrated in a more comprehensive modeling framework that is currently developed within the CHESS project, which combines MDE philosophy with component based development techniques. We described the role of the transformation workflow within CHESS, and we detailed all the transformation steps required to obtain the metrics of interest and to back-annotate the results. A key aspect of the transformation process is the presence of an intermediate

model that allows to improve the reusability and flexibility of the model transformation process. We have described relevant parts of this intermediate model showing its capability in representing dependability properties of the system. Then we detailed the first part of the transformation process, from CHESS ML to the intermediate model, which is the core of the whole transformation process as it selects and organize all the required dependability-related system information.

Next short-term activities concern the finalization of the whole set of transformation steps composing the workflow, and then its implementation within the CHESS framework. Solutions to mitigate the complexity of the resulting dependability analysis model will be inspected. State-space explosion is a well-known issue for state-based modeling, even for models that are built manually, and it may lead to huge computational costs both concerning solution time and memory requirements. Several approaches to cope with this problem exist, grouped in largeness avoidance and largeness tolerance techniques [22]. An investigation of the most suitable approaches to cope with the largeness of the generated models will help to improve the overall efficiency and applicability of the proposed approach.

Moreover, although originally developed for the analysis of embedded systems, we are currently studying if and to which extent the intermediate model and the transformation workflow can be profitably used for the analysis of large-scale complex critical infrastructures, as those identified within the ongoing PRIN "DOTS-LCCI" project [23]. To cope with the huge system complexity, e.g. in terms of number components, we are specifically investigating how to couple the transformation process with large avoidance techniques based on decomposition/aggregation ([24]).

ACKNOWLEDGMENT

This work has been partially supported by the European Project ARTEMIS-JU-100022 CHESS [3] and by the Italian Ministry for Education, University, and Research (MIUR) in the framework of the Project of National Research Interest (PRIN) "DOTS-LCCI: Dependable Off-The-Shelf based middleware systems for Large-scale Complex Critical Infrastructures".

REFERENCES

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

[2] Object Management Group, "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms", version 1.0, February 2008. http://www.omg.org/spec/QFTP/1.0/.

[3] ARTEMIS-JU-100022 CHESS - Composition with guarantees for High-integrity Embedded Software components aSsembly. http://www.chess-project.org.

[4] A. D'Ambrogio, G. Iazeolla, R. Mirandola. "A method for the prediction of software reliability", In Proc. of the 6th IASTED Software Engineering and Applications Conference (SEA'02), 2002.

[5] V. Cortellessa, H. Singh, B. Cukic. "Early reliability assessment of UML based software models", In WOSP '02: Proceedings of the 3rd international workshop on Software and performance, pages 302–309, New York, NY, USA, 2002. ACM.

[6] M. Kovacs, P. Lollini, I. Majzik, A. Bondavalli. "An integrated framework for the dependability evaluation of distributed mobile applications", In SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems, pages 29–38, New York, NY, USA, 2008. ACM.

[7] A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza and G. Savoia, "Dependability Analysis in the Early Phases of UML Based System Design", International Journal of Computer Systems – Science & Engineering, Vol. 16 (5), Sep 2001, pp. 265-275.

[8] M. Dal Cin, G. Huszerl, K. Kosmidis, "Evaluation of Safety-Critical Systems Based on Guarded Statecharts", In A. Williams, editor, Proc. Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE'99), IEEE Computer Society Press, 1999.

[9] G. Huszerl, I. Majzik, A. Pataricza, K. Kosmidis, M. Dal Cin, "Quantitative Analysis of UML Statechart Models of Dependable Systems", The Computer Journal, Vol 45(3), May 2002, pp. 260-277.

[10] G. Huszerl, I. Majzik, "Modeling and Analysis of Redundancy Management in Distributed Object-Oriented Systems by Using UML Statecharts", In: Proc. of the 27th Euromicro Conference, pp. 200-207., Warsaw, Poland, 4-6. September 2001.

[11] M. Walter, C. Trinitis, W. Karl, "OpenSESAME: An Intuitive Dependability Modeling Environment Supporting Inter-Component Dependencies", In Proc. of the 2001 Pacific Rim Int. Symposium on Dependable Computing, pp 76-84, IEEE Computer Society, 2001.

[12] A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia. "Dependability analysis in the early phases of UML based system design". Journal of Computer Systems Science and Engineering, 16(5):265-275, 2001.

[13] PRIDE – Ambiente di PRogettazione Integrato per sistemi DEpendable, Transformations for Dependability Analysis, Deliverable 2.1, February 2003.

[14] S. Bernardi, J. Merseguer, D. Petriu, "A dependability profile within MARTE", Journal of Software and Systems Modeling, pages 1–24, 2009.

[15] Object Management Group, "A UML profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems", version 1.0, November 2009. http://www.omg.org/spec/MARTE/1.0/.

[16] Object Management Group, "MDA Guide", Version 1.0.1, June 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf.

[17] L. Montecchi, P. Lollini, A. Bondavalli, "Dependability Concerns in Model-Driven Engineering", 2nd IEEE International Workshop on Object/component/service-oriented Real-time Networked Ultra-dependable Systems (WORNUS 2011), March 28-31, 2011. Newport Beach, CA, USA, in press.

[18] MDT/Papyrus. Eclipse Model Development Tools (MDT). http://wiki.eclipse.org/MDT/Papyrus-Proposal

[19] J. Gray. Why do Computers Stop and What Can be Done About it? In Proc. of 5th Symposium on Reliability in Distributed Software and Database Systems, pages 3-12, January 1986.

[20] L.M. Hillah, E. Kindler, F. Kordon, L. Petrucci and N. Trèves, "The Petri Net Markup Language and ISO/IEC 15909-2", CPN Workshop 2009.

[21] L. Montecchi, P. Lollini, A. Bondavalli, "An Intermediate Dependability Model for state-based dependability analysis", Technical Report rcl101115 v2.0, University of Florence, Dip. Sistemi Informatica, RCL group, December 2010.

[22] D. M. Nicol, W. H. Sanders and K. S. Trivedi, "Model-based Evaluation: From Dependability to Security", IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1, pp 48-65, 2004.

[23] PRIN, Programmi di ricerca scientifica di rilevante interesse nazionale – Progetto di ricerca DOTS-LCCI: Dependable Off-The-Shelf based middleware systems for Large-scale Complex Critical Infrastructures, 2008. http://dots-lcci.prin.dis.unina.it/

[24] P. Lollini, A. Bondavalli, F. Di Giandomenico, "A decomposition-based modeling framework for complex systems", In IEEE Transactions on Reliability, Volume 58, Issue 1, pp. 20-33, 2009.