# Dependability Concerns in Model-Driven Engineering

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

*Abstract*—**Model-Driven engineering (MDE) aims to elevate models in the engineering process to a central role in the specification, design, integration, validation, and operation of a system. MDE is becoming a widely used approach within the dependability domain: the system, together with its main dependability-related characteristics, is represented by engineering language models, while automatic transformations are used to generate the analysis models for the dependability analyses. This paper discusses the dependability concerns that should be captured by engineering languages for dependability analysis. It motivates and defines a conceptual model where the specific dependability aspects related to specific dependability analyses can be consistently and unambiguously merged, also detailing the part of the conceptual model supporting state-based dependability analysis methods. Then, it introduces a new intermediate dependability model that acts as a bridge between the high-level engineering language and the low-level dependability analysis formalism, and we discuss its features and its expressive power showing its application for the modelling of a simple but representative case-study.**

***Keywords-engineering languages; conceptual model; intermediate dependability model; dependability analysis; state-based methods***

## I. INTRODUCTION

Model-driven engineering refers to the systematic use of models as primary artefacts throughout the engineering lifecycle [19]. Engineering languages like UML, BPEL, AADL, etc., allow not only a reasonable unambiguous specification and design but also serve as the input for subsequent development steps like code generation, formal verification, and testing. One of the core technologies supporting model-driven engineering is model transformation [20]. Transformations can be used to refine models, apply design patterns, and project design models to various mathematical analysis domains in a precise and automated way.

In recent years, model-driven engineering approaches have been also extensively used for the analysis of the extra-functional properties of the systems (like reliability and safety). To this purpose, language extensions (like the UML profiles for QoS and fault tolerance) were introduced and utilized 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 artefacts in the engineering model. Then, automated tools (mainly based on pattern matching and model transformation) are used to assemble in a modular way 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 address specific dependability aspects that are related to specific analysis objectives. What is actually missing is a more comprehensive discussion on the extra-functional concerns that should be captured at engineering language level for dependability analysis.

This paper aims to address this issue providing the following contributions:

- Identification of the basic dependability-related requirements for engineering languages, i.e., the core set of capabilities that an engineering language should have to support the dependability analysis of software and hardware systems.
- Definition of a conceptual framework for dependability analysis, which is the (semantic) space where the different dependability concerns thought for the different analysis objectives should be homogeneously placed. It is the space where the different dependability concepts required by different analysis methods and objectives are merged, avoiding duplications and inconsistencies. The resulting conceptual model should constitute the basis for the construction of a comprehensive UML profile for dependability analysis, which is currently under development within the CHESS project [1].
- Specification of the part of the conceptual framework supporting state-based dependability analysis methods, which are methods widely used for dependability and performance quantitative assessment.

Another important issue in model-driven engineering concerns the reusability and flexibility of the model transformation process, which can be improved defining an intermediate model that acts as a bridge between the high-level modelling language and the low-level dependability analysis formalism. Focusing on this research direction and basing on previous works adopting this type of transformation process, the paper provides the following additional contributions:

- Definition of the logical structure of a new Intermediate Dependability Model (IDM), which encompasses the dependability features belonging to the conceptual model that are required for the state-based analysis.

- Application of the new IDM for the modelling of a simple but representative case-study, showing its capability and expressiveness in capturing the required system dependability features.

The structure of the paper is the following. Related works concerning model-driven engineering approaches for dependability analysis are discussed in Section II. Section III provides an overview of the CHESS project, also discussing the role of the conceptual model and of the intermediate dependability model. Section IV outlines the main dependability-related requirements that an engineering language should meet to support dependability analysis. The conceptual model is then detailed in Section V. Section VI provides an overview of the intermediate dependability model, while the case-study is presented in Section VII. Final conclusions are drawn in Section VIII.

## II. Related Works

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 in more abstract languages like UML.

The idea of translating UML models to dependability models was elaborated and refined in several papers. In [9], Markov chains are used to model the reliability of middleware architectures described in extended UML. In [14] a fault tree of the system is derived processing a set of UML diagrams; the authors of [15] derived dynamic fault trees model from UML model extended with specific attributes. In [16] UML models are enriched with probability values, and the system's failure is evaluated using Bayesian rules. The work in [17] defines a framework for the evaluation of distributed systems, where the analysis model is derived from an overall model composed of an UML model and a network topology description.

Structural UML diagrams form the basis of a transformation to Timed Petri Net dependability models in [10]. Performability and dependability models are instead constructed on the basis of behavioral UML diagrams in [11]. 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 [12] where a transformation from statechart diagrams to Stochastic Reward Nets is presented. In a hierarchical modeling approach, this behavioral level transformation can be used effectively to construct the sub-models of redundancy managers whose behavior determines replica management and service restoration (recovery) [13]. Tools that implement transformation approaches have been also developed; as an example, the OpenSESAME tool [3] 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 (High Level Design Environment for Dependability) project [2] addressed these issues using an intermediate dependability model, which acts as a bridge between the high-level modeling language and the dependability analysis formalism. Such approach has then been later refined within the PRIDE project [8].

The intermediate model introduces an additional abstraction layer, through a representation that is independent of both the engineering modeling language and the analysis formalism. Albeit 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. As example, in PRIDE Stochastic Activity Networks (SAN) has been used as analysis formalism, while in HIDE, which used a very similar intermediate model, Generalized Stochastic Activity Networks (GSPN) were used. In addition, if we consider $n$ engineering languages and $m$ analysis formalisms, $n \times m$ possible transformations between them exist; however, if using an intermediate model, only $n + m$ transformation rules are enough to cover all the possible combinations.

Despite several approaches propose model transformations for dependability analysis, still there is not a common understanding of what are the dependability properties that should be included in a high-level modeling language, and how such properties should be organized. Indeed, the lack of support for dependability attributes, and extra-functional attributes in general, is one of the most recognized weaknesses of UML-based languages in the design of critical and embedded systems. Some attempts to address this issue have been carried out recently: the QoS&FT profile [5] provides support to the modeling of Quality of Service concepts and supports the description of fault-tolerant architectures based on object replications; the MARTE profile [6] provides a framework for the modeling of real-time properties of embedded systems. More focused on dependability, the work in [7] defines the DAM profile, an UML profile for dependability analysis based on MARTE. The attributes included in DAM are defined merging in a single profile all the attributes used in different works surveyed from the literature, with the objective to create a single dependability modelling language. It allows the user to define a wide range of attributes, but the strategy adopted for its definition has produced a quite intricate language where the user is allowed to introduce inconsistencies and ambiguities in the model, not having any guidance or constraints to help him.

A possible way to address this issue is to have a semantic space where the dependability concepts required for the different analysis can be homogeneously merged. The main challenge is to ensure that no inconsistencies or redundancies exist between the information that supports different analysis techniques. 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

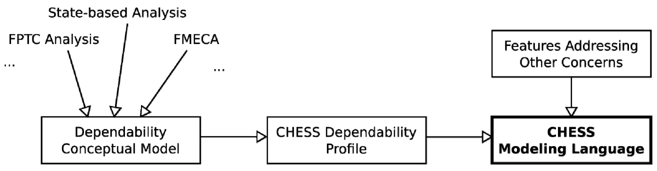
The CHESS project aims at developing, applying and assessing an industrial-quality Model-Driven Engineering (MDE) infrastructure for the specification, analysis and verification of extra-functional properties (predictability, dependability and security) in component-based systems modelling. The methodology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

The scientific and technical objectives of CHESS can be then summarized as follows:

1. Building modelling languages that support extra-functional properties in the specification of component-based systems.
2. Developing tools for the evaluation of extra-functional properties to support the design and development of component-based systems.
3. Adapting component infrastructures for the integration of real-time and dependable patterns. Real-time and dependable systems have associated specific design patterns; these patterns must be integrated in the software infrastructures, the modelling tools and the software generators.
4. Validating the approach with innovative component based case studies.

The CHESS philosophy refers to a particular MDE initiative, the Model-Driven Architecture (MDA) defined by OMG [18]. In the workflow promoted by the MDA the system designer creates a Platform Independent Model (PIM), which is independent of the execution platform that will actually implement the system. From the PIM model enriched with deployment information a Platform Specific Model (PSM) is then generated by automated transformations. From the 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 saves the user from having 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 high-level CHESS model that has triggered the analysis.


State-based Analysis
FPTC Analysis
FMECA
...
...
Features Addressing Other Concerns
Dependability Conceptual Model
CHESS Dependability Profile
CHESS Modeling Language
Figure 1. Definition of the CHESS Modeling Language and the role of the dependability conceptual model.

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 like UML, SysML and MARTE. The definition of such language is performed as an iterative process in which the language is constantly refined and harmonized, addressing inconsistencies and clashes between different domains of interest.

For what concerns 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 a set of system dependability attributes, and some of them may be shared between different methods. For example, the concept of "failure" is shared between all the analysis methods mentioned above. The conceptual model merges the concepts needed by each analysis method into a single semantic space (Fig. 1). Such concepts are then instantiated into the CHESS Dependability Profile, which allows to enrich a CHESS ML model with information related to dependability and safety. The CHESS Dependability Profile, as well as the features addressing other concerns (e.g., schedulability) will contribute to the final specification of the CHESS ML language. 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.

Among the techniques for dependability analysis introduced above, 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 the system's state and its possible changes with respect to time, allowing representing more complex interaction between components than using simpler combinatorial formalisms like fault-trees. Dependability-related measures are evaluated assessing the probabil-

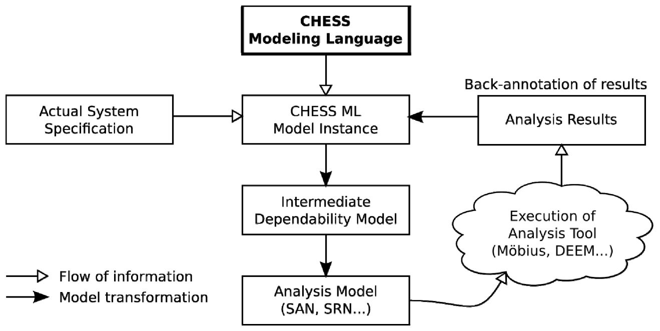

CHESS Modeling Language
Back-annotation of results
Actual System Specification
CHESS ML Model Instance
Analysis Results
Intermediate Dependability Model
Execution of Analysis Tool (Möbius, DEEM...)
Flow of information
Model transformation
Analysis Model (SAN, SRN...)

Figure 2. CHESS workflow for state-based dependability analysis.

ity of the system of being in a certain state; performance-oriented measures can be evaluated enriching the model with costs and rewards.

The workflow for state-based dependability analysis uses a three-step transformation process (see Fig. 2). The system is modeled using the CHESS ML language, based on the system's specification. In the first step, from the CHESS ML model instance that represents the system to be analyzed, an Intermediate Dependability Model (IDM) representation is derived through model transformation; the second transformation step generates the "implementation" of the analysis model in a specific formalism starting from the IDM representation. The analysis model is then used as input for the analysis tool, which generates a set of analysis results as output. The third transformation step back-annotates the results in the original CHESS ML model; such results can then be used as input for subsequent analyses, thus allowing an incremental refinement of the system's model.

## IV. Dependability-related Modeling Requirements

The definition of the CHESS Dependability Profile starts from the identification of dependability aspects/ features/ characteristics/ properties that a UML profile should address to support dependability analysis of software and hardware systems.

In this section we discuss the dependability requirements that should be addressed in the profile. The list we provide here is the synthesis of requirements coming from industrial companies, integrated with the experiences of the research partners within the CHESS Consortium. Other works in the literature have managed to identify modeling requirements for dependability analysis, with the objective to define UML extensions. To the best of our knowledge, one of the most comprehensive works on this topic is [7]; the requirements defined in CHESS are covering the aspects identified in such work as well.

The requirements we have identified are the following:

R1 *Need to model the structure of a system (as a composition of subsystems/components).*
Systems are often organized hierarchically, where higher-level components are decomposable in lower-level components; e.g., an electronic control unit is composed of a processor, a memory and other devices. Moreover, system functionalities may be seen as logical components that depend on a set of subcomponents. The health conditions of higher-level components depend on the state of its subcomponents.

R2 *Need to define the different types of components, like stateful (e.g., memory, CPU) or stateless (e.g., buses), hardware (e.g., processing elements, memory items) or software (e.g., wrappers, schedulers).*
Components can be classified with respect to different characteristics, which may result in different behaviors with respect to dependability properties. For example, components can be classified with respect to the presence of an internal state: stateful elements have internal state and may develop errors; conversely stateless errors do not have an internal state.

R3 *Need to define the common types of fault-tolerant structures (e.g., k-of-n, TMR, etc.) and the role that each component plays in such structures (e.g., if a component is a voter, a spare, a variant, etc.).*
Fault-tolerance structures usually provide an effective way to contrast threats that affect system's components. Many different structures exist, some implementing complex fault-tolerance techniques. Proper definition of such structures at UML level allows automatically deriving a dependability model that represents the details of that specific kind of fault-tolerant structure.

R4 *Need to model dependency relations between components (to define the error propagation paths).*
Components within the system interacts each other and such interactions may generate propagation paths between them. E.g., a software component depends on the hardware it is running on, and a service may depend on the output produced by another service.

R5 *Need to define the different types of faults (e.g., permanent or transient), errors (e.g., latent or detected) and failures (e.g., minor or catastrophic).*
Different kinds of threats may affect the system's components, having different impact on dependability properties. Different kinds of threats define different types of incorrect behaviour of components.

R6 *Need to define specific dependability properties and attributes to qualify the components (e.g., failures distributions, error detection coverage, error latency, MTTF, MTTR, MTBF, duration/frequency of the maintenance activities).*
Different components are characterized by different dependability properties, which specify the behavior of components with respect to faults, errors, failures and repairs.

R7 *Need to represent the different maintenance policies (corrective/preventive) and maintenance activities (repair, replace, overhaul).*
In repairable systems, maintenance policies and activities specify how existing threats are removed from the system. Comparing different maintenance policies allows evaluating tradeoffs between availability and reliability of the system.

R8 *Need to define metrics of interests (e.g., availability, reliability, fault-tree relevant metrics, criticality level, N Failure Tolerance, requirements coverage).* Since the dependability properties in the system's model are mainly used for dependability analysis, it should be possible to specify the objective of the analysis.

## V. THE CONCEPTUAL MODEL

We now introduce the conceptual model for the representation of dependability properties in component based architectures. The CHESS Modeling Language (CHESS ML), which is currently under development within the project, will provide an implementation of this conceptual model into a concrete UML-based representation. A specific subset of the resulting language, the CHESS Dependability Profile, will provide the required support for the representation of dependability concepts.

The goal in the definition of a conceptual model is to identify at abstract level the main concepts that belong to the domain of interest, together with their relationships. The conceptual model is explicitly chosen to be independent of design or implementation concerns. At this stage, how the elements that have been identified will be represented in a UML dependability profile has not been addressed yet.

This conceptual model takes into account for the different dependability analysis methods that are supported in the CHESS methodology and it identifies the concepts that need to be included in the description of component based architectures to support the analysis techniques. It consists of several logical packages, each one targeting a specific aspect related to dependability and safety analysis. The packages have been logically grouped in four abstraction levels, each one corresponding to a different step in the construction of the analysis model:

- *Components*. This level identifies the basic components of the system and their dependability and safety properties. In such component-based context, the extensions introduced at this level are used to enrich the basic hardware and software components with information related to dependability and safety.
- *Threats*. This level of abstraction concerns with the identification of the threats that may affect the system and its dependability and safety properties.
- *Means for dependability*. At this level the means to attain dependability are identified and described. From a general perspective, these are solutions and countermeasures which are developed to deal with the threats identified in the above level and to reach the dependability and/or safety requirements.
- *Analysis*. This level of abstraction addresses both the identification of analysis' objectives and the notation to support specific analysis methods. Each analysis method requires specific extensions to support the automatic/semiautomatic generation of analysis models and (possibly) back-annotation into the original model.

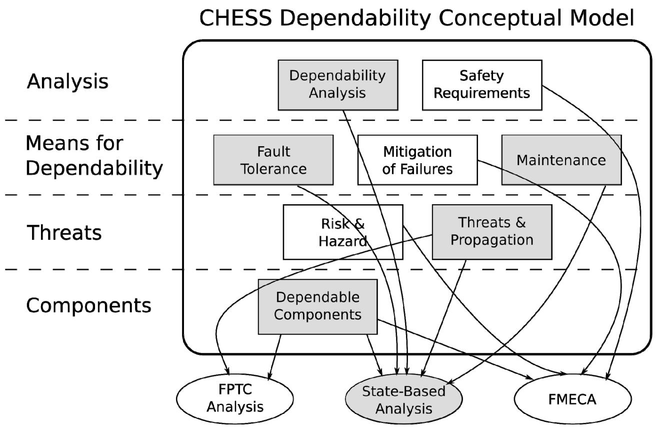


Figure 3. Logical structure of the CHESS Dependability Conceptual model.

The four levels are populated by different packages, which address the concerns related to that specific level in different ways (see Fig. 3). Different analysis techniques rely on different packages, but they can share common packages as well. For example, qualitative attributes related to risk may not be useful for state-based analysis, while quantitative attributes related to error propagation may not be useful for FMECA analysis. However, the two methods may rely on the same definition of hardware or software component.

In the rest of this paper we focus on state-based dependability analysis and the related logical packages are detailed in the following subsections.

### *A. Dependable Components*

This package is used to describe the system structure (requirement A1) with respect to dependability analysis, and to describe the different types of components (requirements A2 and B2). This package allows also defining the basic relations between components (requirement A4).

The conceptual elements that are addressed by this package are the following:

- **Component**. Components are the basic blocks of the system (hardware or software) with respect to dependability analysis. From the dependability point of view, a component may be affected by faults, errors, and failures (see Section V.B). Different kinds of components may exist. Further classifications of components with respect to dependability analysis are listed in the following.
- **Hardware / Software.** Components of a system may be hardware or software. Software components are affected by transient faults only. Indeed, we assume that permanent faults in software components have been removed by debugging and testing activities prior to their release. Hardware components are instead subject to both permanent and transient faults.
- **Stateful / Stateless.** Components may have an internal state; in this case they are *stateful*, otherwise *stateless*. Stateful and stateless components have different behaviors with respect to error propagation. Stateful components, having an internal state, may

develop internal errors. In such components a failure occurs when an error reaches the service interface. Conversely, stateless components do not have an internal state, and any fault that may occur immediately leads to a failure of the component.

- **Use relations.** Components are connected through their interface, and they communicate in order to implement the system's function. Components may require interfaces to perform their function (i.e., they are "clients"), or they may provide interfaces to the other components (i.e., they are "servers"). When a component uses the functionality provided by another component, error propagation may occur. This kind of relationships are then necessary to derive basic error propagation paths between interacting components. Use relations may be enriched with a time delay and a probability that error propagation actually takes place.
- **Composition relations.** Components may be composed of subcomponents, with multiple levels of depth. With respect to dependability analysis composition relations describe how failures of subcomponents propagate to the higher-level component. When it is not specified otherwise, it may be assumed the composed component is fully functional if all of its subcomponents are fully functional.

## B. *Threats & Propagation*

This package allows to define different types of fault, errors and failures (requirement B1) and complex dependency relations between components (requirement A4). Simple error propagation paths are derived from the "ComponentUses" and "ComponentIsComposedOf" relations within the Dependable Components package.

The conceptual elements that are addressed by this package are the following:

- **Fault.** A fault that may affect a component of the system, possibly generating errors and/or failures. With respect to dependability analysis we distinguish two kinds of faults. Internal faults develop inside components with a certain occurrence rate or delay, and they may be permanent or transient with a certain probability. For software components the probability of faults of being permanent is constrained to be zero. External faults are generated by external causes and they may originate from failures of other components. Components may be affected by different faults, having different effects on component's health.
- **Error.** An error is a deviation from correct system's state. Different errors may develop inside a component as a result of propagation or activation of latent faults. If compensation occurs, after a certain amount of time errors may disappear from the internal state of the component. For example, an erroneous value stored in a memory cell gets compensated if it is overwritten by a correct value, before the wrong one is actually used by other components.
- **Failure mode.** When an error reaches the service interface of the component, a failure occurs. The failure of the component may take different forms, called failure modes. Different failure modes may affect a component, as result of different error propagation paths. Failure modes are characterized by their domain, detectability, consistency, and consequences [4]. Failures may propagate in different ways to other components depending on their failure mode.
- **Propagation path.** These relations identify propagation path that exists in the system. Propagation paths are characterized by a time delay and a probability that the propagation actually occurs. A weight may also be specified, that represents the probability of that path with respect to the others. Propagation paths connect faults to errors, errors to failure modes, or failure modes to (external) faults. Moreover, propagation may also take place between errors and errors, since existing errors may generate additional errors in the same component.

## C. *Fault Tolerance*

This package concerns the specification of how faults and errors are handled by the system. This aspect includes the description of redundancy structures (requirement A3), and the description of error detection mechanisms (e.g., online tests, see requirement B1).

We foresee two ways to describe a fault tolerant structure in the system: i) by marking a specific component of the system as being fault tolerant, and associating to them a redundancy scheme and the related attributes; ii) by identifying existing components as being part of the implementation of a redundancy structure; in this case each component is enriched with additional attributes, e.g., the role it plays within the structure. These two ways serve to different purposes in the design process. In the former case the "internals" of the fault tolerant structure are not visible to the user, making this approach better suited to compare different choices in the early design phases. In the latter, the redundancy structure is built starting from pre-existing model elements, making this approach better suited for the analysis of already finalized system architectures.

The conceptual elements that are addressed by this package are the following:

- **Fault tolerance structure.** In the first case components which are implemented by a fault tolerant structure are identified by specific model elements. The attributes that are attached to such elements further characterize the fault tolerant structure by specifying the redundancy scheme that it implements, as well as its parameters.
- **Redundancy manager.** This conceptual element is used to specify that a component in the system has the role of redundancy manager of a fault tolerant structure. The redundancy manager is the interface of the structure to the other components and it is characterized by the redundancy scheme that it implements. This element is used to provide the second

of the two methods for the definition of fault tolerant structures.

- **Adjudicator.** This element is used to specify that a component in the system has the role of adjudicator in a fault tolerant structure. The adjudicator checks the correctness of variants operation. This model element is used to provide the second of the two methods for the definition of fault tolerant structures.
- **Variant.** This element is used to specify that a component in the system has the role of a variant in a fault tolerant structure. This model element is used to provide the second of the two methods for the definition of fault tolerant structures.
- **Detection activity.** Error detection activities are used to detect the presence of errors in stateful components. Such activities are characterized by their coverage and false alarm ratio. Coverage is the probability to detect an error, given that it is present; false alarm ratio is the probability to detect an error, given that it is not present. Additional attributes specify when the activity should be executed, its duration, which components are tested and the kind of errors that the activity can detect.
- **Performer of detection activity.** Detection activities may be performed by other components of the system. In such cases, relations must exist in the dependability model between the detection activity and the component which is in charge of executing it. With respect to dependability analysis, such relation can be used to take into account the effect of not being able to perform the activity because of the failure the component that executes it.

### D. *Maintenance*

This package is used to describe maintenance related properties and policies (requirement C1). Maintenance can be preventive or corrective. Preventive maintenance is performed on the system according to a previously settled time scheduled program; corrective maintenance is performed only when needed. The need for corrective maintenance is determined by some monitoring activities which are executed on the system.

The conceptual elements that are addressed by this package are the following:

- **Repair activity.** Components may be repaired during the lifetime of the system. Repair activities may be planned or they may be performed when some event occurs in the system (e.g., the failure of a component). Repair activities may have a duration, or they may be assumed instantaneous, if their duration is negligible with respect to the duration of other events in the system. It is possible that repair activities do not always complete successfully, but instead that they are successful with a given probability.
- **Replace activity.** Components may be replaced during the lifetime of the system. Often replace can be seen as a special case of repair, and it is then characterized by the same attributes: a duration, a probability of success and a specification of when the activity should be performed. However, it is possible that a component is replaced with a different one. Given that the new component must be functionally interchangeable with the replaced one, it may still have different dependability properties. It is the case, for example, of a software component that is upgraded to a new version. For this reason replace activities may specify the properties of the component used as replacement.
- **Overhaul activity.** Overhaul is a maintenance activity that has as its primary objective to prevent components to reach an age in which their dependability attributes and/or performance are degraded by frequent failures. In components having an increasing failure rate, overhaul tries to keep the failure rate below a certain threshold.
- **Performer of maintenance activity.** Maintenance activities may be performed by other components of the system. This may be the case, for example, of automated recovery activities (e.g., restart of a software application). In such cases, relations must exist in the dependability model between the maintenance activity and the component which is in charge of executing it. In a similar way to detection activities, such relation can be used to take into account the effect of not being able to perform the activity because of the failure the component that executes it.

### E. *Dependability Analysis*

The modeling elements included in this package allow the definition of analysis objectives (requirement D1). These elements are heavily dependent on the kind of analysis that should be performed on the system. To specify the objectives of dependability analysis it should be specified at least: i) the type of measure(s) that should be evaluated (e.g., reliability, availability...); ii) the type of evaluation that should be performed on such measures (e.g., instant of time, steady-state...); iii) the time point(s) on which the measures should be evaluated, in case of transient analysis; and iv) the component with respect to which the measures should be evaluated. The full specification of the metrics of interest may require additional information (e.g., related to the statistical properties of results), which is however more related to the solution process.

The conceptual elements that are addressed by this package are the following:

- **Availability.** Availability is defined as the readiness for correct service. An availability measure is characterized by the type of evaluation that should be performed for that measure, which can be an instant of time, interval of time or steady-state evaluation. In order to track the dependability requirements, it should also be possible to specify a minimum and maximum value that the measure that it is allowed for the measure.
- **Reliability.** Reliability is defined as the continuity of correct service. In a similar way as above, a reliability measure is characterized by the type of evaluation

that should be performed and, possibly, its minimum and/or maximum value.

- **Safety.** Safety is a measure of continuous safeness, or equivalently, of the time to catastrophic failure. Safety is thus reliability with respect to catastrophic failures. In a similar way as above, a safety measure is characterized by the type of evaluation that should be performed and, possibly, its minimum and/or maximum value.
- **Target of measure.** This relation connects a dependability measure to the target for which it should be evaluated. In fact, to completely specify the dependability measures of interest, the conceptual elements defined above (reliability, availability, safety) should be related to some component of the system. In addition, it should be possible to connect a dependability measure to some specific failure mode of a component. In this case the measure is evaluated only with respect to the specified failure modes. This can be useful for example when components have multiple failure modes, but a part of them are considered benign failures or they are not interesting for dependability analysis.
- **Steady-state.** One of the possible evaluation types for dependability measures is steady-state evaluation.
- **Instant of time.** One of the possible evaluation types for dependability measures is instant of time evaluation. This element is characterized by an attribute that specifies the actual instant of time at which the measure should be evaluated.
- **Interval of time.** One of the possible evaluation types for dependability measures is interval of time. This element is characterized by two attributes that specify the boundaries of the interval of time in which the measure should be evaluated.

## VI. THE INTERMEDIATE DEPENDABILITY MODEL

The conceptual model defined in Section V has been mapped to an Intermediate Dependability Model (IDM) to be used as support for state-based dependability analysis. Due to limited space we do not describe the whole IDM metamodel here; a full and detailed definition of the metamodel is provided in [22]. Rather, in this section we provide some insights on the expressive power the intermediate model can offer, while keeping a representation that is independent of both the high-level modeling language (e.g., UML), and the analysis formalism (e.g., GSPN).

With respect to the state of the art, our intermediate model introduces several new modeling features, which enable the modeling of: i) detailed faults/errors/failures propagation chains; ii) components having multiple failure modes; iii) internal error propagation and possible errors compensation; iv) preventive and corrective maintenance; v) error detection activities; and vi) different types of measures of interest.

The intermediate model for state-based dependability analysis is organized in five logical packages, which are closely related to the packages of the conceptual model. The "Statistics" package contains a collection of most common probability distributions, and it can be extended to include additional ones. The "Dependable Components" package defines the basic components of the system, and maps the concepts of the same package in the conceptual model. "Threats & Propagation" maps the same package of the conceptual model, and the part of package "Fault Tolerance" which relates to redundancy structures. The other part of such package, which addressed monitoring concerns, is grouped with maintenance elements in the "Maintenance & Monitoring" package. Finally, the "Dependability Analysis" package maps the concepts related to the definition of the objective of the analysis.

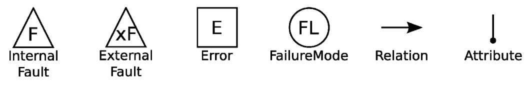


Figure 4. The graphical notation used for the main elements of the IDM.

It is worthwhile to note that not all the elements in the conceptual model have a direct representation in the intermediate model: some of them (like the redundancy structures), are actually represented as a composition of more elementary IDM elements (composite components and logical conditions on propagation paths). This approach has been chosen in order to keep the intermediate metamodel as simple as possible and avoid duplicated notation. The definition of transformations to the analysis model would otherwise be more complex and prone to inconsistencies.

The IDM representation is composed of *nodes* and *relations*, and it can be conveniently expressed using a graphical notation (Fig. 4). In the IDM graphical notation, 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.

Please note that the main purpose of the graphical notation is to help understanding how model elements are organized within IDM and which are the relations between them. As stressed above, the intermediate model is generated by automated transformations, and the user should not be able to modify or even access it. In the following section we give some insights on the modeling power of the IDM with a simple but representative case study.

## VII. CASE STUDY: FIRE DETECTION SYSTEM

The system that has been considered is a fire detection system mounted on-board of automatic light train systems. In such environment, both safety and reliability are of utmost importance for the service provider. On one hand, a reliable detection of fire events must be provided, to ensure the safety of passengers; on the other hand a transport system should provide a continuous service, and false alarm should then be avoided.

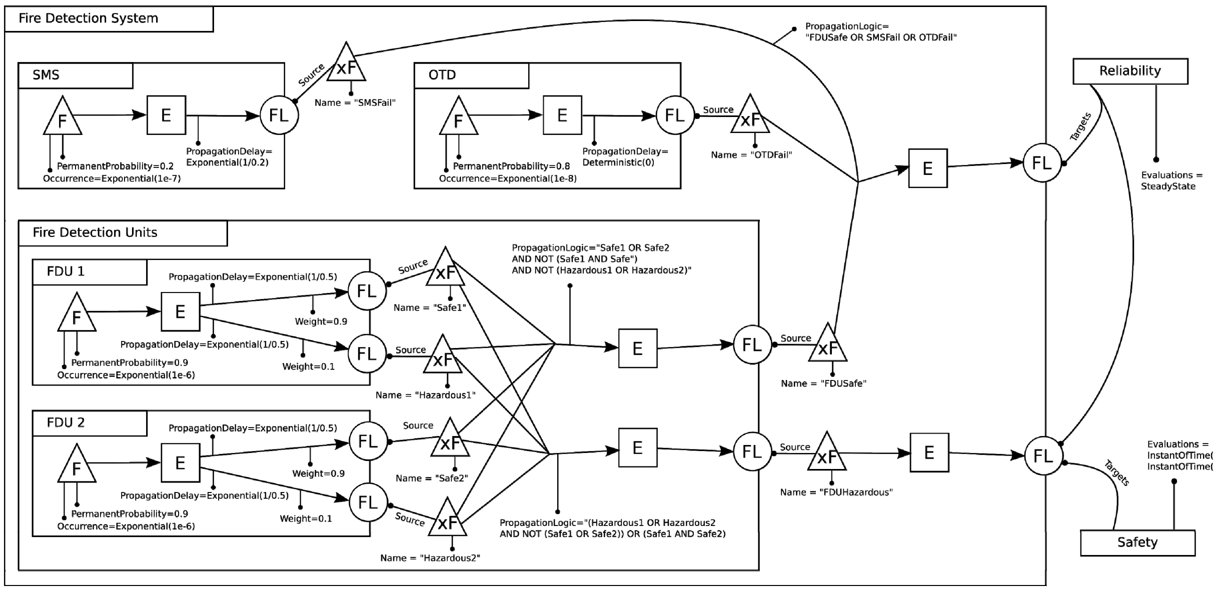


Figure 5. IDM representation of the fire detection system.

The fire detection system takes its decision based on a set of Smoke Sensors (SMS) and a set of Over-Temperature Detectors (OTD). These sensors are managed by two Fire Detection Units (FDU), which analyze the data received from them and trigger the alarm signal when a fire event is detected. Both the FDU are able to detect a fire event, but only one of them is allowed to control the sensors at the same time.

Each FDU is subject to two failure modes: a "safe" failure mode, and a "hazardous" failure mode. When a FDU fails in the safe mode, an alarm is triggered and the other FDU takes control of the sensors. When instead a FDU fails in the hazardous mode, it prevents the other from having access to the sensors' data, thus making the system unable to detect a fire event. A system hazard occurs if: i) at least one FDU fails in a hazardous mode, or ii) if both the FDU fail, in any of the two failure modes. For what concerns sensors, their failures are always considered safe.

As mentioned above, this kind of system has both reliability and safety requirements. As such, we are interested in two kinds of measures: the reliability of the system at steady-state, which can be expressed with its Mean Time To Failure (MTTF); and the probability that hazardous event has not occurred at different instants of time.

## A. IDM model of the system

The IDM representation of the fire detection system described above is shown in Fig. 5. For simplicity, the smoke sensors have been considered as a single hardware component; the same holds for over-temperature detectors.

The component corresponding to the set of smoke sensors is depicted in the upper-left part of the figure (labelled "SMS"); after a certain delay ("Occurrence" attribute) a fault develops inside the block, and with a certain probability it may be transient or permanent ("PermanentProbability" attribute). The fault generates an error in the component, and after an additional propagation delay ("PropagationDelay" attribute) the error reaches the external interface of the component, causing a failure of the smoke sensors block. The component corresponding to over-temperature detectors is labelled "OTD" in the figure and its structure is similar to the one used for smoke sensors. However, temperature sensors are stateless component (they do not have an internal state) and any fault immediately causes them to fail. To represent this aspect in the intermediate model, the propagation delay has been set to zero for the "OTD" component. More in detail, the attribute "PropagationDelay" has been set to follow a deterministic distribution having value zero.

The two FDU are shown in the lower-left part of the figure. As for the model of sensors described above, each of the two FDU may be affected by a fault, which may then generate an error inside the component. In the components corresponding to the two FDUs, an error may cause two distinct failure modes, which correspond to the "safe" and "hazardous" failure modes of the units. The relation that connects the error with each failure mode has two attributes: a "PropagationDelay" that specifies the delay between the generation of the error and the occurrence of the failure, and a "Weight", which determines the relative probability of the two different propagation paths.

The two FDU are enclosed in the higher-level logical component labelled "Fire Detection Units"; the "FailureMode" elements of the single FDUs are connected to "ExternalFailure" elements of the higher-level component, through the "Source" attribute. The higher-level component is then affected by four different faults: the safe failure of FDU 1, the hazardous failure of FDU 1, the safe failure of FDU 2, and the hazardous failure of FDU 2. Different combinations of these failures (specified by the "PropagationLogic" attribute of propagation relations) propagate as two different failure modes for the higher-level component: a "safe" and a "hazardous" failure mode.

The highest-level component (labelled "Fire Detection System") represents the whole system and is affected by four different kind of faults corresponding to the failures of its subcomponents: i) the failure of the smoke sensors block, ii) the failure of the over-temperature detectors block, iii) the "safe" failure of the FDUs block, and iv) the "hazardous" failure of the FDUs block. When one of the first three events occurs, it propagates as a safe failure mode of the system-level component. The "hazardous" failure mode of the FDUs block propagates as a "hazardous" failure of the system.

The measures of interest are specified by the two "Reliability" and "Safety" model elements, which are connected to the "FailureMode" elements of the component representing the whole system. The "Evaluations" attribute specifies the type of measure that should be evaluated: "SteadyState" for reliability and "InstantOfTime" for safety.

## VIII. Conclusions

This paper discussed discussed some of the main dependability concerns in model-driven engineering. Specifically, it addressed the following (strictly related) topics: i) the definition of a semantic space where the dependability concepts required by different analysis methods can coexist; ii) the specification of the dependability concepts required for state-based analyses; iii) the definition of a new intermediate dependability model for state-based analyses; iv) the modelling of a case-study.

Besides embedded systems, we are currently inspecting the opportunity to extend and refine the conceptual model, the intermediate dependability model and the state-based transformation workflow for the modelling and evaluation of large-scale complex critical infrastructures, as those identified within the ongoing PRIN "DOTS-LCCI" project [21]. In order to manage or mitigate the huge system complexity, we are exploring the possibility to combine the model-driven approach for the modeling part with specific decomposition/aggregation approaches for the solution process ([23]).

## Acknowledgment

This work has been partially supported by the European Project ARTEMIS-JU-100022 CHESS [1] 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" [21].

## References

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

[2] A. Bondavalli, M. Dal Cin, D. Latella, and A. Pataricza, "High-level Integrated Design Environment for Dependabiliy (HIDE)". In Proc. WORDS'99, 1999.

[3] M. Walter, C. Trinitis, and 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.

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing", IEEE Transactions on Dependable and Secure Computing, pp. 11-33, January-March, 2004

[5] 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/.

[6] 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/.

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

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

[9] V. Issarny, C. Kloukinas, and A. Zarras, "Systematic Aid for Developing Middleware Architectures", In Communications of the ACM, Issue on Adaptive Middleware, Vol 45(6), pp 53-58, June 2002

[10] 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.

[11] M. Dal Cin, G. Huszerl, and 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.

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

[13] G. Huszerl and 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.

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

[15] G. J. Pai and J. B. Dugan. "Automatic synthesis of dynamic fault trees from UML system models", In Proc. of the 13th International Symposium on Software Reliability Engineering (ISSRE 2002), pages 243–254, 2002.

[16] V. Cortellessa, H. Singh, and 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.

[17] M. Kovacs, P. Lollini, I. Majzik, and 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.

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

[19] D. C. Schmidt, "Model-Driven Engineering", IEEE Computer 39 (2), February 2006.

[20] K. Czarnecki and S. Helsen, "Classification of Model Transformation Approaches", In Proc. of OOPSLA'03, Anaheim, California, USA, 2003.

[21] 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.

[22] L. Montecchi, P. Lollini, and A. Bondavalli, "An Intermediate Dependability Modeling for state-based dependability analysis", Technical Report rcl101115, University of Florence, Dip. Sistemi Informatica, RCL group, November 2010.

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