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. An intermediate model eases the definition of transformations, providing an additional abstraction layer, through a representation that is independent both of the high-level modeling language and of the specific analysis formalism. In this work we define a new Intermediate Dependability Model (IDM) to support state-based dependability analysis. With respect to previous approaches based on intermediate models we introduce new modeling features; more in detail, we provide support for the modeling of detailed fault/error/failure chains inside component, complex propagation paths, internal error propagation and error compensation, multiple failure modes, error detection activities, maintenance activities, and a more detailed specification of the measures of interest that should be evaluated. In order to provide a better understanding of how IDM modeling elements are used we also provide some simple modeling examples using a graphical representation. Finally, model transformation rules are provided in order to automatically derive an IDM model starting from models built using a prototype of the CHESS ML language, an engineering modeling language based on UML developed in the ongoing CHESS project.

KEY WORDS
Intermediate Dependability Model - Model-Driven Engineering - Component-Based Development - Dependability Analysis - State-Based Methods - UML - Model Transformations - CHESS ML.

1 Introduction
The advancements in information technologies in recent decades has brought a continuing growth in the complexity and size of computer systems. The need to cope with such complexity has produced new design and development techniques, among which Component-Based Development (CBD) and Model-Driven Engineering have been particularly successful. In
Component-Based Development the system is built by assembling reusable components having well-specified interfaces; in Model-Driven Engineering the system is designed abstracting from its implementation, using high-level models that provide an abstract solution of the problem.

Model-based assessment is one of the most powerful fault-forecasting techniques. In such approach a model of the system is built using some formalism, and then it is analyzed to obtain useful insights and measures concerning the system’s dependability properties. Most simple dependability models are based on combinatorial approaches (e.g., fault trees), in which each component of the system is considered independent of the others. State-based techniques use more advanced modeling formalisms (e.g., Stochastic Petri Nets and Markov Chains) that allow to represent complex interactions and dependencies between system’s components. State-based models are then solved, either analytically or by simulation, in order to evaluate the measures which are of interest.

The MDE philosophy promotes automated transformations for the generation of different kinds of “artifacts” starting from the system’s specification in a high-level engineering language (e.g., UML [21] or SysML [20]). In the last years this concept has been extended also to dependability analysis: several approaches that derive a dependability analysis model starting from an architectural description of the system in some high-level language. Most of the work present in literature define a direct transformation from the engineering model to the analysis model; however, it has been shown how the use of an intermediate model is highly desirable when defining such transformations.

In this work we define a new intermediate model to support state-based dependability analysis. With respect to previous approaches based on intermediate models we introduce new modeling features; more in detail, we provide support for the modeling of detailed fault/error/failure chains inside component, complex propagation paths, internal error propagation and error compensation, multiple failure modes, error detection activities, maintenance activities, and a more detailed specification of the measures of interest that should be evaluated.

Section 2 briefly describes the context of this work, which is centered the Model-Driven Engineering (MDE) philosophy, and introduces some related works. The Intermediate Dependability Model metamodel is provided in 3 while some modeling examples are provided in Section 4. Conclusions are then drawn in 7.

# 2 Dependability analysis in model-driven engineering

Model-driven engineering refers to the systematic use of models as primary artefacts throughout the engineering lifecycle [24]. 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 [10]. 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.

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 16, Markov chains are used to model the reliability of middleware architectures described in extended UML. In 12 a fault tree of the system is derived processing a set of UML diagrams; the authors of 22 derived dynamic fault trees model from UML model extended with specific attributes. In 9 UML models are enriched with probability values, and the systems 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 5. 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 14 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) 15. Tools that implement transformation approaches have been also developed; as an example, the OpenSESAME tool 25 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 \[13\] 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 \[23\]. 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.

3 The Intermediate Dependability Model

In this section we provide the definition of the metamodel of the new Intermediate Dependability Model (IDM), also including some usage examples that show how it can be used to model dependability properties of systems.

The aim that has been pursued in the definition of this intermediate model is to define a flexible but structured language to model dependability properties of systems. The approach we follow is similar to the one adopted within the HIDE project \[13\] and refined later within the PRIDE project \[23\], in which an intermediate dependability model to support the dependability analysis of UML model were defined. Based on such works, we define a new intermediate model, which provides additional modeling features and an improved modeling power. More in detail, the new features that have been introduced in our new intermediate model 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.

Figure 1. Convenience in using an intermediate model. Number of transformation to be defined in order to support \(n\) high-level modeling languages and \(m\) analysis formalisms.
The elements of the IDM metamodel are grouped in five logical packages: Statistics, Dependable Components, Threats & Propagation, Maintenance & Monitoring and Dependability Analysis (Figure 2). The Statistics package defines some of the most common probability distributions to be used in the model to describe quantities in a probabilistic way (e.g., time delays related to the occurrence of events in the system). Dependable Components defines the basic elements of the system and their attributes; Threats & Propagation allows to model the faults/errors/failure chain within a single component and between different components. The Maintenance & Monitoring package includes the elements to model maintenance and monitoring activities, while Dependability Analysis allow to define in details the measures of interest that should be evaluated, and the type of evaluation that should be performed. In the following we provide a detailed description of the IDM metamodel. The overall metamodel is depicted in Figure 3 using the UML Class Diagram notation.

The elements in the IDM metamodel and their attributes are described in detail in the following subsections, grouped by package. For each element in the metamodel we provide a description of the concept that it represents and a list of attributes that are attached to it. Each attribute is described as a triple (attribute, type, cardinality), specified using the following notation:

- Attribute \{type\} \{cardinality\}

The cardinality of \(n\) specifies that the related attribute has a cardinality of \(n\), two natural numbers separated by dots, \(n..m\), specify that the cardinality of the related number must be between \(n\) and \(m\). The star (*) is a placeholder for any natural number greater than or equal to zero. An attribute having cardinality \{*\} is therefore optional, and an upper bound to its cardinality is not specified. An attribute having cardinality \{1..*\} does not have an upper bound as well, but it must be specified at least once.

Some elements are defined as specializations of other elements of the metamodel. This concept is similar to the UML “generalization” concept: a generalization is a relation between two elements that specifies that one of them (the child, or subtype) should be considered as a specialization of the other one (the parent, or supertype). When two elements are connected by a generalization relation, the child element inherits all the attributes attached to the parent, but it may also have additional ones. In the UML graphical notation generalization relations are represented by a line that connects the two elements, with an empty triangle on the side of the parent element. Another concept that is related to the generalization relation is the concept of abstract elements: similarly to abstract classes in object-oriented programming, an abstract element may have attributes, but it may not be directly instantiated. Abstract elements allow to define hierarchies of elements in a structured way.

The metamodel that is defined in this section uses basic datatypes to specify the attributes of model elements. More in detail, the “String” and the “RealNumber” datatypes are used to represent string of characters and real numbers, respectively. The definition of these basic datatypes is not included in the definition of the intermediate model, which instead aims to capture the aspects strictly related to dependability analysis. In order to manage the basic datatypes we assume that is is possible to reuse some already existing representation, e.g., the representation provided by the transformation tool, or the representation provided by the high-level modeling language.
Figure 3. The IDM metamodel, described using the UML class diagram notation.
3.1 Statistics package

This package is a library that allows to describe probability distributions and associate them to the attributes of other elements in the model. Some of the most used probability distributions in dependability analysis are defined in this package. However, the following list should not be considered exhaustive: if needed, other distributions may be included as well, as a specialization of the abstract element Distribution.

- **Distribution.** This is an abstract element that represents a generic probability distribution. It is the supertype of all the other distributions and it is used as type for the attributes that may follow any probability distribution.

  - **Abstract**

- **Exponential.** This element represents the exponential probability distribution, which is characterized by the following PDF and CDF:

  \[
  f(x) = \begin{cases} 
  \lambda e^{-\lambda x} & x \geq 0, \\
  0 & x < 0,
  \end{cases} 
  \]

  \[
  F(x) = \begin{cases} 
  1 - e^{-\lambda x} & x \geq 0, \\
  0 & x < 0.
  \end{cases}
  \]

  - Specialization of **Distribution**

  - **Rate** \{RealNumber\} \{1\}. The parameter \( \lambda > 0 \), known as rate.

In dependability analysis models, random faults of components as well as repair delays are often modeled using the exponential distribution.

- **Deterministic.** This element represents the degenerate, or deterministic, probability distribution. The degenerate distribution is localized at a point \( x_0 \) and the PDF and CDF are the following:

  \[
  f(x) = \begin{cases} 
  1 & x = x_0, \\
  0 & \text{otherwise},
  \end{cases} 
  \]

  \[
  F(x) = \begin{cases} 
  0 & x < x_0, \\
  1 & x \geq x_0.
  \end{cases}
  \]

  - Specialization of **Distribution**

  - **Value** \{RealNumber\} \{1\}. The value \( x_0 \) where the distribution is localized.

This particular probability distribution allows to treat numeric constants using the probability theory. More in detail, in this intermediate model it allows to assign numeric constants to attributes that require a probability distribution.

- **Gaussian.** This element represents the normal (or Gaussian) probability distribution, which is characterized by the following PDF and CDF:

  \[
  f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},
  \]

  \[
  F(x) = \frac{1}{2} \left[ 1 + \text{erf} \left( \frac{x-\mu}{\sqrt{2\sigma^2}} \right) \right].
  \]

  - Specialization of **Distribution**

  - **Mean** \{RealNumber\} \{1\}. The parameter \( \mu \), known as mean.

  - **Variance** \{RealNumber\} \{1\}. The parameter \( \sigma^2 \geq 0 \), corresponding to the variance.

The normal probability distribution is often employed to model data that have been obtained experimentally.

- **Uniform.** This element represents the uniform probability distribution, which is characterized by the following PDF and CDF:

  \[
  f(x) = \begin{cases} 
  \frac{1}{b-a} & x \in [a,b], \\
  0 & \text{otherwise},
  \end{cases} 
  \]

  \[
  F(x) = \begin{cases} 
  0 & x < a, \\
  \frac{x-a}{b-a} & x \in [a,b], \\
  1 & x \geq b.
  \end{cases}
  \]

  - Specialization of **Distribution**

  - **Lower** \{RealNumber\} \{1\}. The parameter \( a \), corresponding to the lower bound of the interval.
The uniform probability distribution is used to model quantities (e.g., amount of time) that fall between two values and have a constant probability density on each point of that interval. In dependability analysis it can be used, for example, to model time delays that are constrained by timeout mechanisms.

- **Gamma.** This element represents the Gamma probability distribution, which is characterized by the following PDF and CDF:

\[
\begin{align*}
    f(x) &= \begin{cases} 
    \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x} & x \geq 0, \\
    0 & x < 0,
    \end{cases} \\
    F(x) &= \begin{cases} 
    \frac{\gamma(\alpha, x)}{\Gamma(\alpha)} & x \geq 0, \\
    0 & x < 0,
    \end{cases}
\end{align*}
\]

with

\[
\Gamma(k) = \int_0^\infty t^{k-1} e^{-t} dt, \quad \gamma(k, x) = \int_0^x t^{k-1} e^{-t} dt.
\]

- **Specialization of Distribution**
  - **Alpha** \{RealNumber\} \{1\}. The parameter \( \alpha > 0 \) of Gamma distribution, known as shape parameter.
  - **Beta** \{RealNumber\} \{1\}. The parameter \( \beta > 0 \) of Gamma distribution, known as scale parameter.

The Gamma distribution is used to model waiting delays in queuing theory. Moreover, the Gamma distribution is a generalization of other common distributions: with \( \alpha = 1 \) it is actually an exponential distribution having rate \( \beta \). In general, if \( \alpha \) is a natural number, the Gamma distribution is actually an Erlang distribution, i.e., the sum of \( \alpha \) independent random variables following an exponential distribution with rate \( \beta \).

- **Weibull.** This element represents the Weibull probability distribution, which is characterized by the following PDF and CDF:

\[
\begin{align*}
    f(x) &= \begin{cases} 
    \left( \frac{\alpha}{\beta} \right)^\alpha \left( \frac{x}{\beta} \right)^{\alpha-1} e^{-(x/\beta)^\alpha} & x \geq 0, \\
    0 & x < 0,
    \end{cases} \\
    F(x) &= \begin{cases} 
    1 - e^{-(x/\beta)^\alpha} & x \geq 0, \\
    0 & x < 0,
    \end{cases}
\end{align*}
\]

- **Specialization of Distribution**
  - **Alpha** \{RealNumber\} \{1\}. The parameter \( \alpha > 0 \) of the Weibull distribution, known as shape parameter.
  - **Beta** \{RealNumber\} \{1\}. The parameter \( \beta > 0 \) of the Weibull distribution, known as scale parameter.

The Weibull distribution is particularly useful to model the time to failure of components whose failure rate is influenced by the age of the component. More in detail, if \( \alpha < 1 \) the rate decreases with time, while if \( \alpha > 1 \) the rate increases with time. When \( \alpha = 1 \) an exponential distribution with rate \( \lambda = 1/\beta \) is obtained.

### 3.2 Dependable Components package

This package defines the elements used to describe the components of which the system is composed and their attributes with respect to dependability. In this context a "component" could be any element of which the system is composed, and it is not necessarily a component of a component-based architecture.

- **Component.** This element represents a generic system’s component.
  - **Name** \{String\} \{1\}. The name, or more in general a string, that identifies the component.
  - **Faults** \{Fault\} \{1..*\}. The faults that are associated with the component. A fault can be associated with multiple Component elements, allowing to model faults that simultaneously affect multiple components, e.g., faults related to the external environment like accidents and natural disasters. Each component must have at least a fault associated to it.
– **Errors** \{Error\} \{1..*\}. The error(s) that is associated with the component. Each component must have at least an error associated to it. Stateless components (i.e., those that do not have an internal state) cannot develop internal errors and any fault causes them immediately to fail. For convenience this kind of components is modeled adding a “logical” error, that completes the usual faults/errors/failure chain and allows to treat all the components in the same way. To model them properly, the propagation delay of this error to the service interface is set to zero, therefore immediately causing a failure.

– **FailureModes** \{FailureMode\} \{1..*\}. The failure modes that are associated with the component. Each component must have at least a failure mode associated to it.

### 3.3 Threats & Propagation package

In this package the elements needed to model “threats” to dependability are provided. It includes the elements needed to model the details of the faults, errors, and failures chain, both within the single components and through the interactions between them.

- **Threat.** This is an abstract element that represents a generic threat, i.e., a fault, an error, or a failure.
  - **Abstract**
  - **Name** \{String\} \{1\}. The name, or more in general a string, that identifies the threat.

- **Fault.** This is an abstract element that represents a generic fault. Faults are then specialized as internal faults, which develop within the boundaries of the component, and external faults, which may be caused by the failure of other system’s components.
  - **Abstract**
  - Specialization of **Threat**

- **InternalFault.** This element represents an internal fault.
  - Specialization of **Fault**
  - **Occurrence** \{Distribution\} \{1\}. The probability distribution of the time of occurrence of the component to which it is associated.
  - **PermanentProbability** \{RealNumber\} \{1\}. The probability that the fault is a permanent fault.
  - **TransientDuration** \{Distribution\} \{0..1\}. The mean duration of a transient fault. If this attribute is not specified it is assumed that transient faults have a duration which is negligible with respect to the other time quantities within the system.

- **ExternalFault.** This element represents an external fault, which may be caused by the failure of other system’s components.
  - Specialization of **Fault**
  - **Source** \{FailureMode\} \{0..1\}. The failure mode that it is perceived as external fault.

- **Error.** This element represent a latent error inside a component.
  - Specialization of **Threat**
  - **VanishingTime** \{Distribution\} \{0..1\}. This is an optional parameter that specifies the probability distribution of the time after which the error disappears from the component’s internal state. 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 read and used by other components. The actual value of this property may depend on several factors, including the frequency of utilization of the component, any error correction mechanisms that may exist and their effectiveness.

- **FailureMode.** This element represents a failure mode of a component. A failure is an event that occurs when the service delivered deviates from the correct service. The deviation may assume different forms, which are called failure modes. Failure modes can be classified based on their domain, detectability, consistency, and consequences [3].
Specialization of Threat

- Domain \{FailureDomain\} \{0..1\}. Characterization of the failure mode with respect to the failure domain.
- Detectability \{FailureDetectability\} \{0..1\}. Characterization of the failure mode with respect to the failure detectability.
- Consistency \{FailureConsistency\} \{0..1\}. Characterization of the failure mode with respect to the failure consistency.
- Consequences \{FailureConsequences\} \{0..1\}. Characterization of the failure mode with respect to the failure consequences.

- **FailureDomain.** An enumeration that is used to specify the domain of a failure. The domain distinguish failures in which the delivered services has incorrect timing from the failures in which the delivered services has incorrect content. If both the timing and the content of the service are incorrect the failure mode is classified as halt, if the state of the component as perceived by the user is constant, or erratic otherwise. The values of which this enumeration is composed of are the following:
  - “Content”. The content of the information delivered at the service interface deviates from the correct implementation of the system’s functions.
  - “Early Timing”. The timing of service delivery or its duration are not correct. More in detail, the service is delivered too early with respect to the system’s specification.
  - “Late Timing”. The timing of service delivery or its duration are not correct. More in detail, the service is delivered too late with respect to the system’s specification.
  - “Halt”. The external state of the system becomes constant: the system activity, if any, it is not perceptible to the users. In this case the service both the timing and the content of the service are incorrect. A particular case of halt is the silence, i.e., no service is delivered.
  - “Erratic”. All the other cases where the delivered service is incorrect both in timing and in content, but the external state of the system is not constant.

- **FailureDetectability.** An enumeration that is used to specify the detectability of a failure. The detectability distinguish between failures that are signaled to the user and those that are not signaled. Signaling of the failure at the service interface may originate form detection mechanisms that verify the correct delivery of the service. The values of which this enumeration is composed of are the following:
  - “Signaled”. The delivery of incorrect service is detected by detection mechanisms within the system and it is signaled to the users of the service.
  - “Unsignaled”. The delivery of incorrect service is not signaled to the users.

The fact that a failure is signaled or not may be an important information in safety analysis of the system. Signaled failures may not be relevant for safety, if the system is able to move to a safe state after the signaling of the failure.

- **FailureConsistency.** An enumeration that is used to specify the consistency of a failure. The consistency describes as a failure is perceived by different users using the same service. The values of which this enumeration is composed of are the following:
  - “Consistent”. The delivery of incorrect service is perceived in the same way by all the users of the service.
  - “Inconsistent”. Some or all the users perceive incorrect service differently. Actually, some users may perceive correct service.

- **FailureConsequences.** An enumeration that is used to specify the consequences of a failure. This dimension describes the possible consequences of the failure on the environment in which the system is operating and classifies the failures according to different levels of severity. The number of these levels and their definition are mostly application-dependent. An example of failure classification with respect to severity is provided by the CENELEC EN 50126 standard for the railway domain [6]:
  - “Insignificant”
Based on the context of application of the system these values can be modified or extended with new values.

- **FaultsExpression.** An element of type *FaultExpression* is a logical expression that specifies how particular combinations of faults may generate errors.
  - Expression {String} {1}. The string that contains the expression.

The grammar of these expressions is as follows:

\[
FT := (FT \text{ and } FT) \mid (FT \text{ or } FT) \mid \text{not } FT \mid <\text{Fault}>
\]

Where *<Fault>* is an element of the intermediate model of type *Fault*.

- **ErrorsExpression.** An element of type *ErrorsExpression* is a logical expression that specifies how particular combination of errors may generate failures.
  - Expression {String} {1}. The string that contains the expression.

The grammar of these expression is as follows:

\[
EE := (EE \text{ and } EE) \mid (EE \text{ or } EE) \mid \text{not } EE \mid <\text{Error}>
\]

Where *<Error>* is an element of the intermediate model of type *Error*.

- **FaultsGenerateErrors.** This relation connects a set of faults with the errors that they may generate inside a component. If multiple errors are specified, we assume that they are all generated at the same time. The faults and the errors must belong to the same component. For the same component multiple relations of this type may be defined.
  - Source {Fault} {1..*}. The faults that are involved in the relation.
  - Destination {Error} {1..*}. The error(s) the may be generated by the faults specified in Source upon their activation.
  - ActivationDelay {Distribution} {1}. Distribution of the delay between the occurrence of the specified faults and their activation, i.e., the instant in which the specified errors are generated.
  - PropagationProbability {RealNumber} {1}. The probability that the propagation of the faults specified in Source to the Errors specified in Destination actually takes place.
  - PropagationLogic {FaultsExpression} {1}. Logical expression that describes how the faults specified in Source must be combined to generate the errors specified in destination.
  - Weight {RealNumber} {0..1}. The weight of this relation with respect to the others that have the same faults as source. If needed, this attribute is used to perform a probabilistic choice between two different propagation paths. If not weight is specified, the default weight is one.

- **InternalPropagation.** This relation represents an error propagation path internal to a component. The errors involved in the relation must belong to the same component. For the same component multiple relations of this type may be defined.
  - Source {Error} {1..*}. The error or the set of errors that generates the propagation.
  - Destination {Error} {1..*}. The error or the set of errors that are generated from propagation.
  - PropagationDelay {Distribution} {1}. Distribution of the time delay after which the propagation takes place.
  - PropagationProbability {RealNumber} {1}. The probability that the propagation of the errors specified in Source to the errors specified in Destination actually takes place.
– **PropagationLogic** {ErrorsExpression} \{1\}. Logical expression that describes how the errors specified in Source must be combined to generate the errors specified in destination.

– **Weight** \{RealNumber\} \{0..1\}. The weight of this relation with respect to the others that have the same faults as source. If needed, this attribute is used to perform a probabilistic choice between two different propagation paths. If not weight is specified, the default weight is one.

Internal propagation is a phenomenon that occurs in some kinds of components and it may accelerate the process for which errors reach the service interface. For example, when a latent error exists in a CPU’s register additional errors may be generated from the computation in the other registers as well.

• **ErrorsProduceFailures** This relation connects a set of errors with the failure mode that they may produce. The errors and failure modes must belong to the same component. For the same component multiple relations of this type may be defined.

  – **Sources** \{Error\} \{1..\*\}. The error or set of errors that causes the failure.

  – **Destination** \{FailureMode\} \{1..\*\}. The failure mode(s) that is involved in the relation.

  – **PropagationDelay** \{Distribution\} \{1\}. Distribution of the time delay after which the propagation takes place. This attribute allows to specify a delay between the generation of an error and the occurrence of a failure. In general, the propagation delay that takes a component from an erroneous state to the failure depends on many factors, including the frequency of utilization of the component, any error correction mechanisms that may exist and their effectiveness. The value of this parameter can be changed in order to perform sensitivity analyses, or it may be evaluated using fault-injection techniques.

  – **PropagationProbability** \{RealNumber\} \{1\}. The probability that the propagation of the errors specified in Source to the errors specified in Destination actually takes place.

  – **PropagationLogic** {ErrorsExpression} \{1\}. Logical expression that describes how the errors specified in Source must be combined to generate the failure mode(s) specified in destination.

  – **Weight** \{RealNumber\} \{0..1\}. The weight of this relation with respect to the others that have the same faults as source. If needed, this attribute is used to perform a probabilistic choice between two different propagation paths. If not weight is specified, the default weight is one.

### 3.4 Maintenance & Monitoring package

The elements contained in this package allow to describe activities that are performed on the system or its components in order to remove (fault removal) or tolerate (fault tolerance) faults. These activities include maintenance activities and monitoring activities. In this context we use the term “monitoring” in its broader sense, meaning all those activities that allow to access the real state of the system.

• **Activity.** This is an abstract element that represents a generic activity that is performed within the system. The activity may be performed by a component of the system, or by some external entity (e.g., a repairman).

  – **Abstract**

    – **Name** \{String\} \{1\}. The name, or more in general a string, that identifies the activity.

    – **Duration** \{Distribution\} \{1\}. Distribution of the amount of time required to execute the activity. In case of activities that are considered instantaneous, this attribute will be set to “Deterministic(0)”, i.e., following a deterministic distribution centered on zero.

    – **When** \{ScheduleExpression\} \{1\}. Expression that it is used to specify when the activity should be executed. The execution policy specified by the expression may include periodic patterns or specific instant of times. Additional conditions on the system’s state may also be specified.

• **ScheduleExpression.** An element of type *ScheduleExpression* is an expression that specifies when an activity should be executed, through a temporal condition and a guard.

  – **Expression** \{String\} \{1\}. The string that contains the expression.
ScheduleExpression expressions are constituted of:

1. A description of the scheduling of the activity with respect to time in the system’s lifetime (T). More in detail, it is possible to specify that the activity is executed only at specific instants of time, or following a periodic pattern.
2. A condition on the system’s state that must hold in order for the activity to be executed (EX). Such condition is expressed through a boolean expression, in which the atomic formulas are predicates on the state of single components.
3. An additional (optional) condition that enables the execution of the activity only in a predefined interval of time (L). If the time interval is specified, the activity is executed only when the other conditions hold within such interval.

The grammar of these expressions is as follows:

\[
S := T [EX] \mid T [EX] \{L\} \\
T := \text{Immediately} \mid \text{AtTime(<RealNumber>)} \mid \text{Periodic(<Distribution>)} \\
EX := (EX \text{ and } EX) \mid (EX \text{ or } EX) \mid \text{not EX} \mid \text{true} \mid \text{FD} \\
FD := \text{Failed(<FailureMode>)} \mid \text{Detected(<Error>)} \\
L := \text{Before(<RealNumber>)} \mid \text{After(<RealNumber>)} \mid \text{Interval(<RealNumber>,<RealNumber>)}
\]

Where the symbols enclosed in angle brackets ¡..¿) are placeholders for any element of the intermediate model having that type. More in detail, the symbols that form the above grammar have the following meaning:

- **Immediately**. The activity is executed immediately as soon as the conditions specified in EX hold.
- **AtTime(<RealNumber>).** The activity is executed at the instant of time specified by the RealNumber element, starting from the beginning of the scenario.
- **Periodic(<Distribution>).** The activity is executed periodically, at intervals of time following the probability distribution that is specified. For periodic activities, we consider the interval of time starting from the beginning of an activity execution and the beginning of the subsequent one. If the activity duration is greater than this interval of time, the activity is executed immediately as soon as the previous execution completes.
- **Failed(<FailureMode>).** Predicate on the state of a component; this predicate is true if the component has failed with the failure mode specified by the FailureMode element.
- **Detected(<Error>).** Predicate on the state of a component; this predicate is true if the error Error has been detected by error detection mechanisms.
- **Before(<RealNumber>).** The activity can be executed only at instants of time prior to the value specified by RealNumber.
- **After(<RealNumber>).** The activity can be executed only at instants of time after the value specified by RealNumber.
- **Interval(<RealNumber>,<RealNumber>).** The activity can be executed only in the interval of time identified by the two RealNumber values. The boundaries are included in the interval.

- **MaintenanceActivity.** This is an abstract element that represents a generic maintenance activity.
  - **Abstract**
  - **Specialization of Activity**
  - **SuccessProbability** {RealNumber} {1}. The probability that the activity is completed successfully.

- **RepairActivity.** This element represents a maintenance activity that involves the repair of a component.
  - **Specialization of MaintenanceActivity**
  - **Target** {Component} {1..*}. The component(s) that is repaired when this activity is performed.
A repair activity reverts a component to its initial state, removing faults, errors and failures that may have developed during its utilization. A repair activity does not need to be a physical operation on an hardware component: restarting a software component can be considered a repair activity as well.

- **ReplaceActivity.** This element represents a maintenance activity that involves the replacement of a component.
  - Specialization of *MaintenanceActivity*
  - Target {Component} {1}. The component that is replaced when the activity is performed.
  - Replacement {Component} {0..1}. It is possible to specify the component that is used as replacement. The new component may have in fact different properties with respect to the one that gets replaced: e.g., the time to fault occurrence may have a different distribution.

Usually the replacement of a component has an effect similar to its repair; however it may have a different effect if the component is subject to aging. When a component is replaced, the component that is used as replacement may be different from the previous one, e.g., software updates or hardware components that are upgraded to higher quality ones.

- **DetectionActivity.** This element is a specific type of activity that represents the execution of an error detection activity.
  - Specialization of *Activity*
  - Coverage {RealNumber} {1}. The conditional probability that an error is detected, given that it has occurred. A detection mechanism provide a judgement on the state of a component, based on how it perceives it. The state of the component may contain an error (E), or it may be error-free (E'); similarly, the detection mechanism may detect an error (D), or it may believe that the component is error-free (D'). When combining the component’s state with the output of the detection mechanism, four possibilities may occur: i) the component state is erroneous and an error is detected (ED); ii) the component state is healthy, but an error is detected (ED); iii) the component state is erroneous, but no error is detected (ED); iv) the component state is healthy and no error is detected (ED).

The *coverage* (or *hit ratio*) of the detection mechanism is defined as:

\[
\text{Coverage} = \frac{ED}{ED + ED'}.
\]

- FalseAlarmRatio {RealNumber} {1}. An other parameter that characterizes detection mechanisms is its false alarm ratio, which is defined as:

\[
\text{False Alarm Ratio} = \frac{ED}{ED + ED'}.
\]

- DetectableErrors {Error} {1..*}. The errors that this activity tries to detect. The errors may also belong to different components.

- **ComponentExecutesActivity.** This relation connects one or more activities with the component that is in charge of executing it. The component that performs the activity does not need to be specified for every activity in the model. When no component has been specified through this relation, we assume that the activity is performed by some external entity (e.g., a repairman).
  - Component {Component} {1}. The component that performs the activity.
  - TargetActivities {Activity} {1..*}. The activities that are executed by the component.

This relation is used take into account the possibility of not being able to perform an activity if the component that should execute it is not working correctly.
3.5 Dependability Analysis package

The elements contained in this package allow to specify the objectives of dependability analysis, that is, the measures of interest that should be evaluated on the system’s model, and how they should be evaluated.

- **DependabilityMeasure**. This is an abstract element that represents a generic dependability measure that should be evaluated on the system’s model.
  
  - **Abstract**
  
  - **Name** {String} {1}. The name, or more in general a string, which identifies the measure of interest.
  
  - **Target** {FailureMode} {1..*}. The failure mode, or the set of failure modes, for which this measure should be evaluated. If the measure should be evaluated for any failure of a specific component, the set of all its failure modes is associated to this attribute.
  
  - **Evaluations** {EvaluationType} {1..*}. This attribute specifies the type of evaluation that should be performed for this measure. It is possible to specify multiple types of evaluation, or the same type with different parameters.
  
  - **RequiredMin** {RealNumber} {0..1}. The minimum value that this measure is allowed to reach, based either on system’s requirements or simply by hypothesis.
  
  - **RequiredMax** {RealNumber} {0..1}. The maximum value that this measure is allowed to reach, based either on system’s requirements or simply by hypothesis.

- **EvaluationType**. This is an abstract element that specifies the type of evaluation that should be performed for a measure.
  
  - **Abstract**

- **SteadyState**. This element is a particular evaluation type and it specifies that the measure should be evaluated at steady-state.
  
  - Specialization of **EvaluationType**

- **InstantOfTime**. This element is a particular evaluation type and it specifies that the measure should be evaluated at instant-of-time.
  
  - Specialization of **EvaluationType**

- **IntervalOfTime**. This element is a particular evaluation type and it specifies that the measure should be evaluated in an interval of time.
  
  - Specialization of **EvaluationType**

- **Reliability**. This element identifies a reliability measure that should be evaluated on the system’s model. The reliability is defined as a “measure of the continuous delivery of correct service” [2].
  
  - Specialization of **DependabilityMeasure**

Based on the types of evaluation that are required for this element (specified through the attribute **Evaluations** of the **DependabilityMeasure** element) different measures will be evaluated on the system:

- If an **InstantOfTime** evaluation is required, the measure \( R(t) \), the reliability at time \( t \), will be evaluated. This measure corresponds to the probability that the system delivers a correct service in the interval \([0, t]\), i.e., that a failure does not occur in such interval of time. If \( X \) is a random variable that governs the time to failure of the system, then:

\[
R(t) = P[X > t] = 1 - P[X \leq t] = 1 - F_X(t),
\]

where \( F_X \) is the CDF of the probability distribution of \( X \).
– If an IntervalOfTime evaluation is required, the interval reliability, \( R(t, T_0) \), will be evaluated. Such measure is defined as the probability that the system is delivering a correct service at the instant of time \( T_0 \) and that it will continue to deliver a correct service for a given time \( t \) \[4\]. When \( T_0 = 0 \) this measure corresponds to the reliability at time \( t \), i.e., \( R(t, 0) = R(t) \).

– If a SteadyState evaluation is required, the steady-state reliability will be evaluated. One possible steady-state measure of the reliability of a system is its Mean Time To Failure (MTTF). If \( X \) is a random variable that governs the time to failure of the system, the MTTF is the mean value \( E[X] \) of such variable.

• Availability. This element identifies an availability measure that should be evaluated on the system’s model. The availability is defined as “a measure of the delivery of correct service with respect to the alternation of correct and incorrect service” \[2\].

– Specialization of DependabilityMeasure

Based on the types of evaluation that are required for this element (specified through the attribute Evaluations of the DependabilityMeasure element) different measures will be evaluated on the system:

– If an InstantOfTime evaluation is required, the measure \( A(t) \), the instantaneous availability at time \( t \), will be evaluated. This measure is defined as the probability that the system is delivering a correct service at time \( t \).

– If an IntervalOfTime evaluation is required, the measure \( A(t_1, t_2) \), the availability in the interval of time \([t_1, t_2]\), will be evaluated. This measure is defined as the portion of time in which the system is delivering a correct service over the time interval \([t_1, t_2]\).

– If a SteadyState evaluation is required, the steady-state availability of the system will be evaluated, i.e., the portion of time in which the system delivers a correct service at steady-state. This measure corresponds to the interval of time availability when the length of the interval approaches the infinity, \( A(0, T) \).

• Safety. This element identifies a safety measure that should be evaluated on the system’s model. The safety is a “is a measure of continuous safeness, or equivalently, of the time to catastrophic failure” \[2\]. Therefore, safety corresponds to reliability with respect to catastrophic failures. When a safety measure is explicitly required, it specifies that only the failure modes having a certain severity level should be taken into account.

– Specialization of DependabilityMeasure

– HazardousLevel \{FailureConsequences\} \{1\}. The minimum severity level for a failure mode to be considered an hazard.

Based on the types of evaluation that are required for this element (specified through the attribute Evaluations of the DependabilityMeasure element) different measures will be evaluated on the system. The measures are the same as for reliability, but they are evaluated with respect only to failure modes that are considered catastrophic.

4 Modeling with the IDM

In this section we provide some modeling examples using the intermediate model defined in the previous section. On one hand we aim to show how the metamodel is able to represent the concepts related to dependability analysis in a detailed way. On the other hand by looking at these examples, which use a graphical representation, it is more straightforward to understand the roles that the different model elements play within an IDM model. Please note that the graphical representation is provided only to allow a better understanding of the model and it can be considered a secondary feature of the intermediate model. Indeed, in the MDE approach, the intermediate model is generated by automatic transformations from the high-level engineering model and the user is not allowed to directly modify it.

The graphical notation that is used for the main elements of the IDM metamodel is shown in Figure 4. Faults are represented by triangular shapes, errors by square shapes, and failure modes by circular shapes. This clear distinction permits an easier understanding of the faults/errors/failure chain and its direction. A relation is represented by an arrow, directed in the same direction as the relation. The attributes of nodes and relations are represented with a line ending with a dot. If an attribute is a reference to another element of the model, the dot is placed near that element. If it is needed, the type of elements that are shown in the model may be shown near the element, enclosed in angled brackets (\( i \ldots i \)).

In the following examples we show only the attributes that are relevant for the modeling of the considered scenarios, while attributes that are secondary (e.g., Name) or optional are omitted to improve the readability of the models.
4.1 Modeling a single component.

Figure 5 depicts the first example, which shows a basic model describing a simple chain of faults, errors, and failures in a component. An InternalFault, an Error, and a FailureMode are associated with the component. After a certain period of time (Occurrence), the fault develops inside the component, and with a given probability it may be transient or permanent (PermanentProbability). After a certain period of time from its occurrence (ActivationDelay) the fault is activated and it generates an error. As an effect of computation the error, after a certain delay (VanishingTime), may be compensated and “absorbed” by the component. After a PropagationDelay has elapsed the error reaches the service interface of the component, causing its failure.

If needed, it is possible to include in the model multiple failure modes for the same component (Figure 6). Each failure mode is represented by a FailureMode element, with the proper attributes, which is added the component. Each failure mode is connected to the errors that generate it through additional ErrorsProduceFailures relations. These relations may have a Weight, which specifies the relative probability of that path with respect to the others.

4.2 Error propagation.

The representation of interactions between components and error propagation between them is of fundamental importance for dependability analysis. Figure 7 shows the intermediate model representation of an error propagation path between two components. Let us assume that Component B uses the service deliverd by Component A: the failure of A may therefore cause the propagation of an error to B.

The single components A and B are modeled as shown in the previous section (Figure 5). An ExternalFault is associated to the failure mode of Component A. In a similar way to internal faults, after a certain delay (ActivationDelay) and with a certain probability (PropagationProbability) the external fault is activated and it generates an error (FaultsGenerateErrors relation) in Component B.
Figure 7. IDM representation of error propagation between two components: the failure of Component A is perceived as an external fault by Component B.

4.3 Internal propagation paths.

If needed it is possible to define more complex propagation paths inside single components. This feature may be useful to model failure that occur only when multiple conditions hold at the same time.

The component depicted in Figure 8 has two internal faults (InternalFault) and three errors (Error) associated with it, which may cause two different failure modes (FailureMode). The two faults are named “Fault_A” and “Fault_B”, while the errors are identified by “Err01”, “Err02”, and “Err03”. When the fault “Fault_A” is activated it generates the error “Err01”, while the activation of “Fault_B” generates the error “Err03”. This behavior is specified through two FaultsGenerateErrors relations. When the state of the component contains an error of type “Err01” with a certain probability and a certain delay it may develop an error of type “Err02”. This aspect is represented through an InternalPropagation relation which involves the two errors and its attributes PropagationProbability and PropagationDelay.

One of the two failure modes is generated by the error “Err02”, with a certain propagation delay. The other one instead requires that component’s state contains both the “Err02” and “Err03” errors. This aspect is represented through an ErrorsProduceFailures relation in which the attribute PropagationLogic is set to the expression “Err02 AND Err03”.

4.4 Detection and maintenance activities.

Figure 9 shows the IDM representation of detection activities and maintenance activities. More in detail, the example involves two components, A and B, with Component A performing an error detection activity on B’s state. The error detection activity is represented by a DetectionActivity element, in which the DetectableErrors attribute is set to the only error associated with Component B. The fact that this activity is executed by Component A is specified through a ComponentExecutesActivity relation connecting Component A with the activity.

The example shows also a RepairActivity that is executed on Component B. In this example the repair of the component is executed as soon as the component becomes failed or when an error has been detected in its state. The attribute When of the repair activity is therefore the following expression:

Immediately [ Detected(Err01) OR Failed(Fail01) ]

In this case the activity is not performed by a component within the system, and therefore no relation of type ComponentExecutesActivity exists. In this case, for example, the activity may model a repair which is performed by an external repairmain.
Figure 8. IDM representation of complex propagation paths inside components.

Figure 9. IDM representation of detection activities and maintenance activities.
4.5 Hierarchical structure

In order to analyze the overall system it is necessary being able to specify how the single elements of the system are composed and grouped to create higher-level components. In the intermediate model composed components are modeled through external faults: when a component is a physical or logical aggregation of subcomponents, the failures of lower-level components are faults of the higher-level component.

Figure 10 depicts the intermediate representation of a component composed of two subcomponents; the composed component fails if at least one of the two subcomponents fails. Lower-level components (A and B) are modeled as in the previous examples; in this figure, however, the names of FaultsGenerateErrors and ErrorsProduceFailure relations have been omitted to save space. Both A and B have a single failure mode, and an ExternalFault of the composed component is associated with each of them. The composed component is therefore affected by two faults, named “FaultA” and “FaultB”. As for atomic components, a FaultsGenerateErrors relation connects the two (external) faults to the error they generate within the component. In this case the PropagationLogic attribute is set to the logical expression “FaultA OR FaultB”. Finally, an ErrorsProduceFailure relation connects this error to a failure mode of the composed component.

This procedure can be further repeated for additional abstraction layers: the failure of the composed component can be considered a fault for some higher-level component and so on, until the composed component is the system itself.

4.6 Measures of interest

The last example concerns with the specification of the measures of interest that have to be evaluated on the system’s model. Let us consider the composite component described in the previous example and, for example, its steady-state availability and instant of time reliability as the objectives of the analysis. For what concerns instant of time reliability, we are interested in its evaluation at three specific instants of time: \( t = 10^2 \), \( t = 10^3 \) and \( t = 10^4 \) hours.

The corresponding IDM model is shown in Figure 11. The availability and reliability measures are represented by the respective Availability and Reliability elements. The failure mode (and thus also the component) with respect to which such measures should be evaluated is specified by the Target attribute, which references the single failure mode of the composed component. The type of evaluation is specified by the EvaluationType attribute, which is set to a SteadyState element for Availability. For Reliability instead such attribute is a set of InstantOfTime elements, corresponding to the three desired instants of time.
The CHESS project [7] 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 development process is supported by different kinds of analyses (e.g., dependability, schedulability), which allow assessing the feasibility of the systems 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 systems 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. The CHESS framework supports and promotes an iterative development process, in which the systems model is constantly updated and refined, based on the results obtained by different analysis techniques. Analysis results are annotated back in the system model, and can be used as input for subsequent analyses. In more advanced stages of development, code generation techniques may 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).

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 [21], SysML [20] and MARTE [19]. 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. Among the different techniques for dependability analysis supported by the CHESS ML language, we focus 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 architectural model is enriched with the needed information, using a subset of the attributes and stereotypes defined in the CHESS Dependability & Security Profile. In addition to the information included in the functional model of the system, specific elements and attributes provide dependability-related information. The main elements used in the (automated) construction of the state-based dependability model in the current version of the CHESS Dependability & Security profile are summarized in Table 1 and are described in the following. A more detailed and up to date description of these entities can be found in CHESS D2.2 [8].

For each element of the profile we provide a brief description of its meaning, the CHESS ML element that it Specializes, its Attributes, and Constraints that may apply for the element, if any.

- **DepComponent**
  - Specializes
    - UML::Component, UML::Property
  - Attributes
    - failure : FailureMode [*]
    - derivedFailure : DerivedFailure [*]
  - Constraints
    - For what concerns *failure* and *derivedFailure* properties, only one of them may have a value at the same time. More in detail, it is allowed to assign a value to *failure* for atomic components, while *derivedFailure* is used for composed components.

This stereotype is used to enrich a component with information related to dependability analysis. In CHESS the hardware platform is modeled using the Hardware Resource Modeling (HRM) package from MARTE. Therefore, the stereotype *DepComponent* may be applied to both hardware and software components. The functional structure of the system is modeled in CHESS using UML a hierarchy of Composite Structure Diagrams, where the higher-level diagram represents the overall system. The generalization relation with UML::Property allows to assign different parameters to different instances of the same component. A subcomponent is in fact considered a “property” (UML::Property) of the higher-level element.

The *failure* or *derivedFailure* attributes may be used to specify multiple failure modes for a component. The attribute *failure* may be used only for atomic components, i.e., those for which a refinement is not defined in the CHESS ML model. The attribute *derivedFailure* may be used only for composed components, i.e., those components whose internal structure is detailed by a Composite Structure Diagram. If it is not specified otherwise, we assume that components may fail with a only a single failure mode.

- **StatefulHardware**
○ Specializes
  – CHESS_ML::DepComponent
○ Attributes
  – faultOccurrence : NFP_Frequency [1]
  – probPermFault : NFP_Real [1]
  – errorLatency : NFP_Duration [1]
  – repairDelay : NFP_Duration [1]
  – transDuration : NFP_Duration [1]
○ Constraints
  – None

This stereotype specializes DepComponent and it is used to identify hardware components that have an internal state (i.e., they are stateful). Hardware components may be affected by both transient and permanent faults; the probability that a fault that affects the component is a permanent fault is given by the attribute probPermFault. The occurrence of a fault within system components is assumed to follow an exponential distribution, having rate specified by the faultOccurrence parameter. The mean duration of a transient fault is specified by the attribute transDuration, and it is assumed to follow an exponential distribution as well. In stateful hardware components the occurrence of a fault does not immediately produce a failure, but it generates an error in the state of the component. The error reaches the service interface of the component after a given time delay, exponentially distributed with mean errorLatency, causing the failure of the component. Once the component has failed, the delay that is required for its repair is assumed to be exponentially distributed as well, with mean specified by the parameter repairDelay.

• StatelessHardware
  ○ Specializes
    – CHESS_ML::DepComponent
  ○ Attributes
    – faultOccurrence : NFP_Frequency [1]
    – probPermFault : NFP_Real [1]
    – repairDelay : NFP_Duration [1]
    – transDuration : NFP_Duration [1]
  ○ Constraints
    – This element may not have internal properties.

This element specializes DepComponent and it is used to identify hardware components that do not have an internal state (i.e., they are stateless). The attributes are the same as for stateful hardware elements, except for errorLatency attribute: since they do not have an internal state, a fault in this kind of components immediately cause a failure of the component. As additional constraint, a stateless component may not have internal properties, which would otherwise constitute the “state” of the component.

• StatefulSoftware
  ○ Specializes
    – CHESS_ML::DepComponent
  ○ Attributes
    – faultOccurrence : NFP_Frequency [1]
    – errorLatency : NFP_Duration [1]
    – repairDelay : NFP_Duration [1]
    – transDuration : NFP_Duration [1]
  ○ Constraints
None

This stereotype specializes DepComponent and it is used to identify software components that have an internal state (i.e., they are stateful). Most software components have an internal state, which is constituted by the set of its internal variables. A software component is assumed to only experience transient faults, since permanent software faults should be removed by debugging and testing activities during its development. Therefore, a stateful software has the same attributes of a stateful hardware component, with the exception of the probPermFault attribute.

- StatelessSoftware
  - Specializes
    - CHESS_ML::DepComponent
  - Attributes
    - faultOccurrence : NFP_Frequency [1]
    - transDuration : NFP_Duration [1]
  - Constraints
    - This element may not have internal properties.

This stereotype specializes DepComponent and it is used to identify software component that do not have an internal state (i.e., they are stateless). A stateless software component has only two attributes, faultOccurrence, that specifies its fault occurrence rate, and transientDuration, that specifies the mean duration of a transient fault. Because it is a software element, a stateless software component is only subject to transient faults; because it is a stateless element, the repair of the component is instantaneous and it is performed as soon as the fault disappears. Therefore the repairDelay attribute is not needed for stateless software components. As additional constraint, a stateless component may not have internal properties, which would otherwise constitute the “state” of the component.

- FailureMode
  - Specializes
    - Nessuna
  - Attributes
    - name : String [1]
    - description : String [1]
    - relProp : NFP_Real [1]
  - Constraints
    - None

This element represents a failure mode of a generic component. The attributes name and description are used to store some information that allow the user to distinguish between the different failure modes of components. The attribute relProp specifies the relative probability that a failure mode occurs with respect to the others that are associated with the same component.

- DerivedFailure
  - Specializes
    - CHESS-ML::FailureMode
  - Attributes
    - propagationLogic : propLogicExpr [1]
  - Constraints
    - None
This element represents a failure mode of a composed component and therefore it may be associated only with components that have subcomponents in the functional model of the system. In addition to attributes associated with FailureMode elements, a DerivedFailure is characterized by a logical expression that specifies how the failure of subcomponents are combined to produce the failure mode of the composed component.

- **Propagation**
  - Specializes
    - UML::Connector, MARTE::Allocation
  - Attributes
    - prob : NFP::Real [1]
    - propDelay : NFP::Duration [1]
  - Constraints
    - None

This stereotype is used to enrich functional connections between components with information related to error propagation. More in detail, this stereotype may be applied to functional connection between ports between components (UML::Connector), both hardware and software. Moreover, thanks to the specialization of MARTE::Allocation, this stereotype may be used to characterize relations that describe allocation of software components on the hardware platform. The information on error propagation is provided by two attributes: a probability that error propagation takes place over the connection identified by the stereotype (prob), and a propagation delay (propDelay) that specifies how long does it take for errors to propagate through that relation.

- **FaultTolerant**
  - Specializes
    - UML::Component
  - Attributes
    - redundancyScheme : RedundancyKind [1]
    - schemeAttributes : VSL::Expression [1]
  - Constraints
    - None

In the CHESS ML Dependability & Security profile 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 first approach is realized through the FaultTolerant stereotype, that is used to identify a component that is implemented by a fault-tolerance structure. The redundancyScheme attribute specifies the redundancy scheme that is implemented by the structure, while schemeAttributes is an expression that specifies the values of the parameters that characterize the redundancy scheme. At the time of writing, a complete definition of the redundnacy schemes allowed in CHESS is under development.

- **RedundancyManager**
  - Specializes
    - UML::Component
  - Attributes
The second approach for the specification of redundancy structures uses this stereotype to identify the element that is the redundancy manager of the structure. The redundancy manager enforces the redundancy scheme and it is the service interface of the structure to the other system components. The attribute redundancyScheme specifies the redundancy scheme that is implemented by the structure. At the time of writing, a complete definition of the redundancy schemes allowed in CHESS is under development.

**Variant**

- Specializes
  - UML::Component
- Attributes
  - None
- Constraints
  - The component must be stereotyped as StatefulHardware, StatelessHardware, StatefulSoftware or StatelessSoftware.

This stereotype is used to identify components that play the role of variants in a fault-tolerance structure. The stereotype Variant may be applied only to elements that are already stereotyped as StatefulHardware, StatelessHardware, StatefulSoftware or StatelessSoftware. The stereotype Variant alone in fact does not provide any information on the behavior of the component with respect to faults, errors, and failures.

**Adjudicator**

- Specializes
  - UML::Component
- Attributes
  - coverage : NFP_Percentage [1]
- Constraints
  - None

This stereotype is used to identify the components that play the role of adjudicators in a fault-tolerance structure. An adjudicator checks that variants are working correctly, and signals possible anomalies that are detected. The attribute coverage specifies the probability that, given that a variant is not working correctly, the adjudicator is able to detect it.

# 6 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, the CHESS ML language 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. The transformations that are described in the following are based on the latest released version of the CHESS ML language at the time of writing, defined in [8]. 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.

## 6.1 Projection of hardware elements

In this section we describe the projection of hardware elements of the CHESS ML model in the IDM representation. The stereotypes involved are StatefulHardware and StatelessHardware.
6.1.1 StatefulHardware

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 attributes faultOccurrence and probPermFault are mapped to the Occurrence and PermanentProbability attributes of the Fault element, respectively. The errorLatency attribute is mapped to thePropagationDelay attribute of an ErrorsProduceFailures relation. Finally, the repair delay is modeled using a RepairActivity, having itsDuration 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 Figure 12 for the case where the component is affected by only a single failure mode. The algorithmic description of this transformation is as follows:

- 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 exp_fo of type Exponential, having exp_fo_RATE equal to the value of the sfh.faultOccurrence parameter in the sfh elements;
  - create an element exp_td of type Exponential, having exp_td_RATE equal to the value of the sfh.transDuration parameter in the sfh elements;
  - create an element exp_el of type Exponential, having exp_el_RATE equal to the inverse of the sfh.errorLatency attribute of the sfh element;
  - create an element ft of type InternalFault, having exp_fo as ft.Occurrence, exp_td as ft.TransientDuration, and sfh.probPermFault as ft.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.ActivationDelay=Deterministic(0), fge.Destination=er and fge.PropagationLogic=ft;
- if sfh does not have any FailureMode associated with it:
  - create a node f of type FailureMode and add it to c.FailureModes;
- otherwise:
for each FailureMode fm that is associated with sfh:
  – create an element f of type FailureMode and add it to c.FailureModes;
  – assign to epf.Weight the value of the attribute fm.relProp;

  • finally:
    – create an element exp.rd of type Exponential, having exp.rd.Rate equal to the inverse of sfh.repairDelay;
    – create an element ra of type RepairActivity, having ra.Target=c and ra.Duration=exp.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.

6.1.2 StatelessHardware

The transformation rules involving StatelessHardware elements are very similar to those for StatefulHardware elements; the main difference is that the PropagationDelay attribute of the ErrorsProduce Failures 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.

The transformation is sketched in Figure 13 for the case where the component is affected by only a single failure mode. The algorithmic description of this transformation is as follows:

  • 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 exp.fo of type Exponential, having exp.fo.Rate equal to the value of the sfh.faultOccurrence parameter in the sfh elements;
    – create an element exp.jd of type Exponential, having exp.jd.Rate equal to the value of the sfh.transDuration parameter in the sfh elements;
    – create an element ft of type InternalFault, having exp.fo as ft.Occurrence, exp.jd as ft.TransientDuration, and sfh.probPermFault as ft.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.ActivationDelay=Deterministic(0), fge.Destination=er and fge.PropagationLogic=ft;
  • if sfh does not have any FailureMode associated with it:
6.2 Projection of software elements

In this section we describe the projection of software elements of the CHESS ML model in the IDM representation. The stereotypes involved are StatefulSoftware and StatelessSoftware.

6.2.1 StatefulSoftware

For what concerns software components, the transformation rules follow again a similar structure as those for stateful hardware components. However, since they are only subject to transient faults, the PermanentProbability attribute of the Fault element is always set to zero.

The transformation is sketched in Figure 14 for the case where the component is affected by only a single failure mode. The algorithmic description of this transformation is as follows:

- For each element sfh of type StatefulHardware in the CHESS ML model:
  - create an element c of type Component in the intermediate model;
  - create a relation epf of type ErrorsProduceFailures, having epf.Source=er, epf.Destination=f, epf.PropagationLogic=er and epf.PropagationDelay=Deterministic(0);
create an element \( \text{exp} \_\text{fo} \) of type \( \text{Exponential} \), having \( \text{exp} \_\text{fo}.\text{Rate} \) equal to the value of the \( \text{sfh}.\text{faultOccurrence} \) parameter in the \( \text{sfh} \) elements;

create an element \( \text{exp} \_\text{td} \) of type \( \text{Exponential} \), having \( \text{exp} \_\text{td}.\text{Rate} \) equal to the value of the \( \text{sfh}.\text{transDuration} \) parameter in the \( \text{sfh} \) elements;

create an element \( \text{exp} \_\text{el} \) of type \( \text{Exponential} \), having \( \text{exp} \_\text{el}.\text{Rate} \) equal to the inverse of the \( \text{sfh}.\text{errorLatency} \) attribute of the \( \text{sfh} \) element;

create an element \( \text{ft} \) of type \( \text{InternalFault} \), having \( \text{exp} \_\text{fo} \) as \( \text{ft}.\text{Occurrence} \), \( \text{exp} \_\text{td} \) as \( \text{ft}.\text{TransientDuration} \), and \( \text{sfh}.\text{probPermFault}=0 \), and add it to \( \text{c}.\text{Faults} \);

create an element \( \text{e} \) of type \( \text{Error} \) and add it to \( \text{c}.\text{Errors} \);

create a relation \( \text{fge} \) of type \( \text{FaultsGenerateErrors} \), having \( \text{fge}.\text{Source}=\text{ft} \), \( \text{fge}.\text{ActivationDelay}={}\text{Deterministic}(0) \), \( \text{fge}.\text{Destination}=\text{er} \) and \( \text{fge}.\text{PropagationLogic}=\text{ft} \);

if \( \text{sfh} \) does not have any \( \text{FailureMode} \) associated with it:

create a node \( \text{f} \) of type \( \text{FailureMode} \) and add it to \( \text{c}.\text{FailureModes} \);

create a relation \( \text{epf} \) of type \( \text{ErrorsProduceFailures} \), having \( \text{epf}.\text{Source}=\text{er} \), \( \text{epf}.\text{Destination}=\text{f} \), \( \text{epf}.\text{PropagationLogic}=\text{er} \) and \( \text{epf}.\text{PropagationDelay}=\text{exp} \_\text{el} \);

otherwise:

for each \( \text{FailureMode} \) \( \text{fm} \) that is associated with \( \text{sfh} \):

create an element \( \text{f} \) of type \( \text{FailureMode} \) and add it to \( \text{c}.\text{FailureModes} \);

create a relation \( \text{epf} \) of type \( \text{ErrorsProduceFailures} \), having \( \text{epf}.\text{Source}=\text{er} \), \( \text{epf}.\text{Destination}=\text{f} \), \( \text{epf}.\text{PropagationDelay}=\text{exp} \_\text{el} \) and \( \text{epf}.\text{PropagationLogic}=\text{er} \);

assign to \( \text{epf}.\text{Weight} \) the value of the attribute \( \text{fm}.\text{relProp} \);

finally:

create an element \( \text{exp} \_\text{rd} \) of type \( \text{Exponential} \), having \( \text{exp} \_\text{rd}.\text{Rate} \) equal to the inverse of \( \text{sfh}.\text{repairDelay} \);

create an element \( \text{ra} \) of type \( \text{RepairActivity} \), having \( \text{ra}.\text{Target}=\text{c} \) and \( \text{ra}.\text{Duration}=\text{exp} \_\text{rd} \);

set the attribute \( \text{ra}.\text{When} \) to the expression: "Immediately [ Failed(f1) OR Failed(f2) OR . . . OR Failed(fn) ]", where each \( \text{fk} \) is an element in \( \text{c}.\text{FailureModes} \).
6.2.2 StatelessSoftware

The projection of stateless software elements is carried out in a similar way as for the other kinds of components. However, in this case the only attribute (besides failure modes) that is associated with the component is faultOccurrence. Both the error propagation and the repair of the component are in fact instantaneous, and therefore the attributes PropagationDelay of the ErrorsProduceFailures relations and Duration of the RepairActivity that are associated with the component are set to the value Deterministic(0). Moreover, since the component is a software element it is affected by transient faults only; for this reason the PermanentProbability attribute of the Fault element is set to zero.

The transformation is sketched in Figure 15 for the case where the component is affected by only a single failure mode. The algorithmic description of this transformation is as follows:

- 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 exp_fo of type Exponential, having exp_fo.Rate equal to the value of the sfh.faultOccurrence parameter in the sfh elements;
  - create an element exp_td of type Exponential, having exp_td.Rate equal to the value of the sfh.transDuration parameter in the sfh elements;
  - create an element ft of type InternalFault, having exp_fo as ft.Occurrence, exp_td as ft.TransientDuration, and sfh.probPermFault=0, 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.ActivationDelay=Deterministic(0), fge.Destination=er and fge.PropagationLogic=ft;

- if sfh does not have any FailureMode 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=er, epf.Destination=f, epf.PropagationLogic=er and epf.PropagationDelay=Deterministic(0);

- otherwise:
  - for each FailureMode fm that is associated with sfh:
    - create an element f of type FailureMode and add it to c.FailureModes;
    - create a relation epf of type ErrorsProduceFailures, having epf.Source=er, epf.Destination=f, epf.PropagationDelay=Deterministic(0) and epf.PropagationLogic=er;
    - assign to epf.Weight the value of the attribute fm.relProp;
  - finally:
    - create an element ra of type RepairActivity, having ra.Target=c and ra.Duration=Deterministic(0);
    - 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.

6.3 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 and the fact that it has been marked with such stereotypes means that its internal structure is not considered relevant for the analysis.

For each composed component in the CHESS ML model that has not been 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 subcomponents. If it is not specified otherwise, 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 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 is as follows:

- For each component parent for which a Composite Structure Diagram that refines its internal structure exists:
  - if the component is stereotyped as StatefulHardware, StatelessHardware, StatefulSoftware or StatelessSoftware then:
    - stop without creating any new element in the intermediate model. In this case the component has been marked as an atomic component, and thus its internal structure is considered not relevant for the dependability analysis. Its projection has been already performed in the projection of atomic components.
  - For each subcomponent child check if a further refinement of its internal structure exists in the CHESS ML model
    - if such refinement exists, first execute the projection of the child component;
  - For each subcomponents child that still does not have a representation in the intermediate model:
    - create an element csub of type Component;
    - create an element esub of type Error, and add it to csub.Errors;
    - create an element fsub of type FailureMode and add it to csub.FailureModes;
    - create a relation epf of type ErrorsProduceFailures, having epf.Source=esub, epf.Destination=fsub, epf.PropagationDelay=Deterministic(0) and epf.PropagationLogic=esub;
  - create an element c of type Component;
- for each FailureMode fchild in the intermediate model that is associated with a subcomponent of parent:
  - if it has not already been projected in the intermediate model, create an element xft of type ExternalFault, having xft.Source=fchild and add it to c.Faults;
  - if parent has attached DerivedFailure elements, for each DerivedFailure dfail:
    - create an element fparent of type FailureMode and add it to c.FailureModes;
    - create an element eparent of type Error and add it to e.Errors;
    - create a relation epf of type ErrorsProduceFailures having epf.Source=eparent, epf.Destination=fparent and epf.PropagationLogic=eparent;
    - create a relation fge of type FaultsGenerateErrors having epf.PropagationDelay=Deterministic(0), fge.Destination=eparent;
    - set the attributes epf.Source and epf.PropagationLogic based on the value of the attribute dfail.propagationLogic;
  - otherwise:
    - create an element fparent of type FailureMode and add it to c.FailureModes;
    - create an element eparent of type Error and add it to e.Errors;
    - create a relation epf of type ErrorsProduceFailures having epf.PropagationDelay=Deterministic(0), epf.Source=eparent, epf.Destination=fparent and epf.PropagationLogic=eparent;
6.4 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.

6.4.1 UML::Connector

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. 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.

The transformation is sketched in Figure 16, while the algorithmic description of this transformation is as follows:

- 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:
  o 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;
  o otherwise:
    – create an element xft of type ExternalFault for each element fm of type DerivedFailure that is associated with the server;
• otherwise:
  – create an element xft of type ExternalFault for each element fm of type DerivedFailure that is associated with the server;
• if the internal structure of client is detailed in the CHESS ML model and its subcomponents have a projection in the intermediate model:
  o 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;
  o otherwise:
    – stop and perform the transformation as if there was a direct connection between server and every subcomponent of client;
• otherwise:
  – create a relation fge of type FaultsGenerateErrors, having fge.Source=xft and fge.PropagationLogic=xft;
  – add to fge.Destination the Error element that is 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 based on the value of the attributes conn.PropagationDelay and conn.PropagationProbability, respectively; otherwise set fge.PropagationDelay=Deterministic(0) and fge.PropagationProbability=1.

6.4.2 MARTE::Allocate

Allocation relations of software in CHESS ML are modeled by UML::Abstraction relations, directed from the software component to the hardware component, and stereotyped with MARTE::Allocate stereotype. Allocation relations indicate a possible error propagation path, directed from the hardware component to the software that is allocated to it. The projection of such relations is similar to the one for UML::Connector relations described in Section 6.4.1, considering that the software components “uses” the hardware component on which it is allocated to.

The projection of such relations creates an ExternalFault element for each FailureMode associated with the hardware component. The ExternalFault is then connected, through a FaultsGenerateErrors relation, to the Error elements associated with the software components that are allocated on the hardware component. 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.

The transformation is sketched in Figure 17, while the algorithmic description of this transformation is as follows:

• For each relation UML::Abstraction stereotyped as MARTE::Allocate directed from a component child to a component parent do the following:
  o if the internal structure of parent is detailed in the CHESS ML model and its subcomponents have a projection in the intermediate model:
    – for each element f of type DerivedFailure that is associated with the intermediate representation of parent create an element xft of type ExternalFault and set xft.Source=f;
**7 Conclusions and future work**

This work provides the definition of a new Intermediate Dependability Model (IDM) for state-based dependability analysis. This model can be used to support automated transformations from high-level engineering languages (e.g., UML) to specific dependability analysis formalisms (e.g., Stochastic Petri Nets). The intermediate model facilitates the definition of transformations and produces more flexible and reusable transformation rules.

With respect to previous approaches based on intermediate models we have introduced new modeling features; more in detail, we provide support for the modeling of detailed fault/error/failure chains inside component, complex propagation paths, internal error propagation and error compensation, multiple failure modes, error detection activities, maintenance activities, and a more detailed specification of the measures of interest that should be evaluated. Some simple modeling examples have also been provided, using a graphical notation. Although they are simple, these examples provide some insights on how the main aspects related to state-based dependability analysis can be captured using the IDM metamodel.

Moreover, we have provided some insights on how the IDM defined in this report will be used in the ongoing CHESS project, providing a set of transformation rules that can be used to automatically derive an IDM model starting from a model of the system in the CHESS ML language, an engineering language derived from a subset of standard languages like UML, MARTE and SysML. Since the CHESS ML model and the related CHESS Dependability & Security profile are not finalized at this time, these transformation are not finalized as well. Future works involving the Intermediate Dependability Model defined in this report is aimed to update and enrich the transformation rules from the CHESS ML language to the IDM,
representation, following the evolution and the lifetime of the CHESS Modeling Language. In the lifetime of the CHESS project, transformation from the IDM model to Stochastic Petri Nets will be defined as well, in order to complete the workflow that enables the automatic analysis of the system starting from a CHESS ML model.

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 [1]. 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 ([18]).

Acknowledgment

This work has been partially supported by the European Project ARTEMIS-JU-100022 CHESS [7] 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” [1].

References


