CHESS Modelling Language
UML/MARTE/SysML profile

Prepared by Intecs
<table>
<thead>
<tr>
<th>Date</th>
<th>Polarsys release</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 July 2015</td>
<td>0.9</td>
<td>First version submitted to Polarsys</td>
</tr>
<tr>
<td>30 June 2016</td>
<td>0.10</td>
<td>Added first set of results from CONCERTO projects (new dependability profile) Added support for Contract based design (allows integration with OCRA).</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

This document addresses the specification of the CHESS Modelling Language (CHESSML) which objective is to allow the definition of platform independent models (PIM), platform specific models (PSM) and analysis models according to the CHESS methodology [ref].

CHESSML is defined as a collection-extension of subsets of standard OMG languages (UML, MARTE, SysML); for its definition modeling features already available from other methodologies and languages (e.g. HRT-UML/RCM) have taken into account.

Figure below gives a conceptual view about the CHESSML dependencies.

![CHESSML dependencies Diagram](image)

In the following sections a brief introduction of the main profile entities is given, also by providing some modelling examples. Then and each part of the profile is explained.

2 AN INTRODUCTION TO THE CHESSML PROFILE

This section provides an introduction to the main parts of the CHESS modelling language (CHESSML), by illustrating some modelling examples. A basic knowledge of...
the UML standard is required.

![UML Diagram](image)

**Figure 2: CHESS model and views predefined structure**

Following figures shows the definition of *Interfaces* and *ComponentTypes*.

**Figure 3: Interfaces**

**Figure 4: ComponentTypes**

Figure 5 shows the definition or provided and required ports for a given ComponentType.

**Figure 5: ComponentType’s provided and required ports**
Figure 6 shows the ComponentImplementations, where each ComponentImplementation realizes a given ComponentType. The ComponentImplementation inherits all the specification from the realized ComponentType, in particular ports with provided/required interfaces and operations; then ComponentImplementations can be used to provide implementation details, and then they can be instantiated and connected together.

**Figure 6: CHESS ComponentImplementation**

Figure 7 shows an example of intra-component binding (as of activity diagram) for the Produce operation implemented by the Producer_impl ComponentImplementation appearing in Figure 6; in particular it is modelled that the implementation of the Produce operation perform a call of Store and then Consume operations through required ports of Producer_impl (the On port field visible in the property editor in Figure 7 allows to specify the required port through which the operation call currently selected applies).
Figure 7: Intra-component binding

Figure 8 shows an example of ComponentImplementation instantiation and connection. The instances (in yellow in the figure) are represented as internal part of a root component, the latter representing the entire SW system under design. Each instance comes with the provided and required ports defined for the typing ComponentImplementation.

Figure 8: CHESS ComponentImplementation instances

Figure 9 shows an example of timing annotation for ComponentImplementation instances, by using MARTE RtSpecification stereotype decorating operations exposed through provided ports:
Figure 9: CHESS timing annotation

Figure 10 shows an example of instance model (tree-like view) derived from the model represented in Figure 9; in particular the instances of the software component are represented under the `SwSystem_instSpec` package.

Figure 10: CHESS Instance Model

Figure 11 shows an example of deployment, so about the platform specification; in this case only one computational resource is defined (in grey in the figure). Moreover the software to hardware component instances allocations are modelled using MARTE `Assign` constructs. Each `Assign` comes with `from` and `to` attributes through which the software instance (from) which is allocated to the hardware instance (to) can be specified.
Figure 11: CHESS deployment model
Figure 12 shows hardware entities and ComponentImplementations enriched with dependability information (about stateful, stateless) useful for state-based analysis (SBA).

Figure 12: Dependability information annotated HW components
Figure 13 shows the State-Based analysis context used to run the state-based analysis.

Figure 13: SBA analysis context
Figure 14 shows how the information about FPTC rules (on the top of the figure) and failure types occurring on top level-input ports (on the left of the bottom figure) can be
provided for ComponentImplementations at type and instance level, to allow FPTC analysis.

Figure 14: FPTC specification

Figure 15 shows how the results of the FPTC analysis about failure types occurring at output-port level are back propagated in the model.

Figure 15: Results of the FPTC analysis

3 CORE PROFILE

3.1 STEREOTYPES

CHESS

Represents a model compliant with the CHESS methodology.
Extension:
UML ::Model

Attributes
None

Associations:
None

Constraints:
[1] May have RequirementView, ComponentView, DeploymentView and AnalysisView as owned members.

CHGaResourcePlatform
Extends the MARTE::GQAM::GaResourcePlatform and applies to InstanceSpecification. Allows to specify an instance specification as resource platform to be considered in GaAnalysisContext.

Extension:
MARTE ::GQAM::GaResourcePlatform, UML::InstanceSpecification

Attributes
None

Associations:
None

Constraints:
None

4 Views Profile
This package adds support for the CHESS views, the latter as defined by the CHESS methodology.

4.1 Stereotypes
Requirement View
Requirement view is used to model system requirements in a CHESS model.

Extension:
Package

Attributes
None

Associations:
None

Constraints:
None.

**Component View**

Component view is used to model data types and software components according to the CHESS component model definition. ComponentView is the result of the application of two sub.views: FunctionalView and ExtraFunctionalView.

**Extension:**

Package

**Attributes**

None

**Associations:**

None

**Constraints:**

[1] Entities allowed to be edited in this view through the FunctionalView are: Package, ComponentType, ComponentImplementation, Realization, ClientServerPort, FlowPort, Property, Operation, Interface, Connector, DataType, InterfaceRealization, Dependency, Enumeration, EnumerationLiteral, InstanceSpecification, Slot, StateMachine, ModeBehavior, Mode, ModeTransition, Configuration, Activity, Interaction. (see Section 5 for further restriction upon these entities).

[2] Entities allowed to be edited in this view through the ExtraFunctionalView are:

- For predictability: CHRtSpecification (see section 6)
- For dependability: the entities described in section 7.

**Deployment View**

Deployment View allows to model the hardware platform and software to hardware allocation. It owns the DependabilityView as sub view.

**Extension:**

Package

**Attributes**

assignList : Assign[0..*]

References all the Assign stereotypes used to model software to hardware component/port instances allocation. CHESS profile does not constraints the kind of entity(s) which can own these Assign.

**Associations:**

None

**Constraints:**

[1] Entities allowed to be edited in this view are:
- the ones defined in ..<TBD>.

- MARTE::Allocate::Assign, to model allocations.

[2] Entities allowed to be edited through the Dependability View are described in section 7. Errore. L'origine riferimento non è stata trovata.

**Real Time Analysis View**
Real Time Analysis View allows to model real-time analysis contexts.

Usage to be defined.

**Extension:**
- Package

**Attributes**
- None

**Associations**
- None

**Constraints:**
- [1] It can be used to work with information concerning the different CHESS predictability analysis only.

**Dependability Analysis View**
Allows to model dependability analysis, e.g. StateBasedAnalysis.

**Extension:**
- Package

**Attributes**
- None

**Associations**
- None

**Constraints:**
- [1] It can be used to work with information concerning the different CHESS dependability analysis only.

## 5 COMPONENT MODEL PROFILE
The component model package defines a set of concepts that can be used to model CHESS software component model.

<To be updated with CONCERTO results>
5.1 ENTITIES

5.1.1 Package

From UML.

Additional Constraints:
- PackageMerge. Not addressed in CHESS ML. Note: merge of packages requires definition of sets of transformations. From SysML specification: “Combining packages that have the same named elements, resulting in merged definitions of the same names, could cause confusion in user models and adds no inherent modeling capability”.

5.1.2 Realization

From UML.

Additional Constraints:
- From ComponentImplementation to ComponentType.

5.1.3 ClientServerPort

From MARTE::GCM.

Additional Constraints:
- [1] A request arriving at a delegating port can have only one delegated port able to handle the request.
- [2] A ClientServerPort can provide (through provInterface attribute) or require interfaces (through reqInterfaces attribute); it cannot provide and require interfaces at the same time.
- [3] Multiplicity has to be 1.
- [4] At instance level required port can have only one connector attached.

5.1.4 FlowPort

From MARTE::GCM.

5.1.5 Property

From UML.

Used to model component attributes and internal parts for composite component.

5.1.6 Operation

From UML.

Operation’s method can be modeled through activity diagram or by using UAL.

Additional Constraints:
5.1.7 Interface
From UML.

5.1.8 Connector
From UML.

Additional Constraints:
Connector maps the connector entity defined in the CHESS component model. Semantic variation point regarding what makes connectable elements compatible needs to be fixed.

[1] It can connect ports only.

5.1.9 DataType
From UML

5.1.10 InterfaceRealization
From UML.
Not mandatory in CHESS.

5.1.11 Dependency
From UML.
It can be used to model the interface required by a ComponentType or ComponentImpl: not mandatory in CHESS.

5.1.12 Enumeration
From UML.

5.1.13 EnumerationLiteral
From UML.

5.1.14 InstanceSpecification
From UML.

5.1.15 Slot
From UML.
5.1.16 StateMachine

From UML.

Additional Constraints:

StateMachine in CHESS ML can only be used to model functional behaviour of ComponentImplementation.

Note: constraints about state machine redefinitions (i.e. how it is possible to apply generalization between state machines) are not provided in the current version of this specification.

DeferredEvent are not supported.

5.1.17 ModeBehavior

From MARTE::CoreElements: represents a dedicated state machine to model operational modes for a component implementation.

Note: not currently supported by model transformations.

5.1.18 Mode

From MARTE::CoreElements: represents a state in a state machine representing an operational mode for a component implementation.

Note: not currently supported by model transformations.

5.1.19 ModeTransition

From MARTE::CoreElements.

Note: not currently supported by model transformations.

5.1.20 Configuration

From MARTE::CoreElements; allows to represent a scenario made of a set of component implementation instances which are available in a given mode.

5.1.21 Activity

From UML.

It is used in CHESS to model operation implementation (i.e. the operation’s method in the UML meta-model), in particular intra-component bindings, i.e. if a given operation invokes a required operation, how many times etc.

The use case is the following:
1) the modeler wants to specify intra-component bindings for an operation Op of a given ComponentImpl C1

2) the modeler creates the Activity diagram in the ComponentView as owned behaviour of C1

3) the modeler sets the activity as the method of the operation Op

4) the modeler uses CallOperationAction to set
   a. the operation called
   b. the required port through which the operation is called

5) The modeler uses initial and final activity to complete the activity diagram

Additional Constraints:

[1] activity has to be owned by a ComponentImplementation.
[2] only action CallOperationAction can be used

5.1.22 Interaction

From UML.

Allows to model collaboration scenarios between component implementation instances through sequence diagrams.

Under evaluation in CHESS.

5.1.23 Assign

From MARTE::Allocate.

It is used in CHESS to model the allocation of component implementation instance to hardware instance.

Additional Constraints:

[1] “from” has to be an InstanceSpecification typed as ComponentImplementation.
[2] “to” has to be an InstanceSpecification typed as hardware component.

5.2 Stereotypes

5.2.1 ComponentType

Maps the component type notion of the CHESS component model.

Extension:
Component
Attributes

None

Associations:

None

Constraints:

[1] It cannot own behaviours.

5.2.2 ComponentImplementation

Maps the component implementation notion of the CHESS component model. It can have requirements associated representing technical budgets.

Extension:

Component

Attributes

language : String [0..1]

OS : String [0..1]

sourceCodeLocation : String [0..*]

Associations:

None

Constraints:

None

To b completed

6 ABOUT MARTE USAGE IN THE CHESSML

This section summarizes the set of MARTE entities which are used in CHESSML and in the views defined by CHESS.

Component model (CHESS SystemView, ComponentView, PlatformSpecificationView)

Operational modes can be defined for the given system under design by using MARTE stereotypes (defined in the CoreElements sub-profile), and in particular the stereotyped ModeBehavior state machine, together with Mode states and ModeTransition transitions. ModeTransition can be added in the state machine in response to an event to model how a given change of mode is activated in the system.
The ModeBehavior state machine can be defined at system, component or hardware level.

**Component model (CHESS ComponentView)**

MARTE GCM\(^1\)::ClientServerPort and FlowPort entities are used in CHESSML to model services and data ports respectively for software components (i.e. CHESSML::ComponentImplementation and CHESSML::ComponentType).

The MARTE HLAM\(^2\)::RtSpecification is used in CHESSML to model the following properties for a given operation owned by a software component (i.e. CHESSML::ComponentImplementation):

- Arrival pattern (i.e., periodic, sporadic) *(occKind attribute)*
- Relative deadline *(relDl attribute)*
- Priority *(priority attribute)*.

In addition, HLAM::RtSpecification is extended in CHESSML (as CHRtSpecification stereotype, see Figure 16) to also allow the specification of the following properties for components operations:

- worst case execution time *(WCET)*,
- protected\unprotected access *(protection stereotype’s field)*,
- ceiling priorities (for protected operations only).

- the set of operations that need to be invoked to allow the operation invocation *(invokedoperationReqForSporadicOcc stereotype’s field)*; for sporadic operations only.

Regarding the aforementioned timing properties, any property value (e.g., the period duration) can be expressed in MARTE by using a complex type which, in addition to the value, has a “mode” attribute which allows specifying the operational mode(s) (as MARTE::CoreElements::Mode) in which the provided values are valid.

The partWithPort relationship available for CHRtSpecification allows to model timing annotation for operations which are provided by a given Property, the latter as instance of component owned by a composite component. The Property and the associated CHRtSpecification can be modelled\visualized through the UML composite structure diagrams.

---

\(^1\) the MARTE Generic Component Model (GCM) sub-profile. The “::” denotes the namespace where the stereotype is defined.

\(^2\) the MARTE High-Level Application Modeling (HLAM) sub-profile
The MARTE HLAM::RtFeature is extended in CHESSML (as CHRtPortSlot stereotype, see Figure 16) to allow application of CHRtSpecification to a given instance of component’s port (Slot in UML).

Figure 16: MARTE HLAM::RtSpecification and HLAM::RtFeature CHESSML extensions
Platform specification (CHESS PlatformView)

The modelling of the platform is allowed in CHESSML by using the stereotypes available in the MARTE Hardware Resource Modeling (HRM) sub-profile. In particular processors can be modelled in CHESS by using the MARTE HRM::HwProcessor entity, while busses connecting processors can be modelled by using the MARTE HRM::HWBus entities.

MARTE HRM::HwProcessor comes with the nbCores attribute (as natural numerical value) which allows to specify the number of cores within the HwProcessor.

MARTE SRM3::MemoryPartition is used in CHESSML to represents a virtual address space which insures that each concurrent resource associated to a specific memory partition can only access and change its own memory space.

Software to hardware allocation (CHESS AllocationView)

3 MARTE Software Resource Modeling (SRM) sub-profile
MARTE Alloc\textsuperscript{4}::Assign is used in CHESSML to model software to hardware component (i.e. HRM::HWProcessor) instances allocation.

MARTE Alloc::Assign can be constrained by the MARTE NFPs\textsuperscript{5}::NFPConstraint construct, where the latter allows specifying the operational mode(s) (as MARTE::CoreElements::Mode) for which the Assign holds.

Alloc::Assign can be constrained with NFPs::NFPConstraint where the latter is used to specify the core of the target processor (HRM::HWProcessor) on which the allocation is performed; the target core is provided as numerical value in the NFPs::NFPConstraint specification, according to the number of cores within the HRM::HwProcessor (nbCores attribute).

**Analysis context (CHESS AnalysisView)**

CHESSML uses the MARTE SAM\textsuperscript{6}::SaAnalysisContext entity to specify the information needed to perform a given analysis.

MARTE GQAM\textsuperscript{7}::GaResourcePlatform is used in CHESSML as logical container for the resources referred by an analysis context. CHESSML extends the GQAM ::GaResourcePlatform (as CHGaResourcePlatform stereotype) in order to be applied to UML Packages (MARTE GaResourcePlatform extends the UML Classifier only). In fact in CHESS the resources referred by an analysis context are component instances which are available in a dedicated UML package.

MARTE GQAM::GaWorkloadBehavior and SAM::SaEndtoEndFlow are used in CHESSML (in the AnalysisView) to refer a sequence diagram modelled in the ComponentView (in detail, GaWorkloadBehavior and SaEndtoEndFlow are applied to a UML activity which has a CallBehaviorAction pointing to the UML Interaction representing the information modelled in the sequence diagram). GaWorkloadBehavior can then be used to attach the referred sequence diagram to an analysis context, the latter used to feed end-to-end response time analysis; SaEndtoEndFlow is used to provide an end-to-end deadline for the behaviour modelled in the sequence diagram.

**PSM (PSMView)**

MARTE SAM::SaExecHost is used in CHESSML to represent at PSM level (in the PSMView) the HRM::HwProcessor modelled in the PlatformView. SAM::SaExecHost allows to specify the running schedulable resources, the latter generated in the PSM by the PIM to PSM CHESS transformation.

\textsuperscript{4} MARTE Allocation Modeling sub-profile  
\textsuperscript{5} MARTE Non-functional Properties Modeling (NFPs) sub-profile  
\textsuperscript{6} Schedulability Analysis Modeling (SAM) sub-profile  
\textsuperscript{7} Generic Quantitative Analysis Modeling (GQAM) sub-profile
MARTE SRM\textsuperscript{8}::SwSchedulableResource is used in CHESSML to represent PSM resources that execute concurrently to other concurrent resource. SRM::SwSchedulableResource comes with a priority field associated.

MARTE SRM::SwMutualExclusionResource and SAM::SaSharedResource are used in CHESSML to model protected resources in the PSMView. SRM::SwMutualExclusionResource allows to specify the type of protection protocol used to access the resource (e.g. priority ceiling).

MARTE SAM::SaStep is used in CHESSML to represent an operation in the PSM having the following properties:

- Execution time (execTime:NFP\_Duration) as specified in the component view
- Blocking time (blockT:NFP\_Duration) (retrieved by the analysis)
- Response time (respT):NFP\_Duration (retrieved by the analysis)
- The list of called operations (subUsage:ResourceUsage [0..*]), as specified in the component model intra-component bindings; this information allows to calculate the blocking time.
- The shared resource where the operation’s behaviour is executed (sharedResource:SaSharedResource [0..1])
- The resource which executes the behaviour (concurRes:SchedulableResource [0..1])

The SAM::SaAnalysisContext modelled in the AnalysisView is mapped to a corresponding SAM::SaAnalysisContex entity in the PSM. The SAM::SaAnalysisContext in the PSM refers the entities of the PSM which are relevant for the analysis, in particular the SAM::SaStep, SRM::SwSchedulableResource and SRM::SwMutualExclusionResource+SAM::SaSharedResource protected resources.

7 \textbf{DEPENDABILITY PROFILE}

This section describes the CHESS modelling language for dependability. The profile is based upon the SafeConcert conceptual dependability model \cite{1} originally presented in CONCERTO Artemis project and reviews the previous dependability profile defined in the context of the CHESS Artemis project.

The diagram of the dependability conceptual model is shown in Figure 17, then the UML profile implementing the aforementioned conceptual model is presented in the following sub-section.

\footnote{\textsuperscript{8} Software Resource Modeling (SRM) sub-profile}
7.1 **STEREOTYPES**

Here we list the dependability profile stereotypes defined as implementation of the conceptual model presented in [1].

The profile is defined here by taking into account the possibility to apply dependability information at type and instance level and the possibility to use the dependability profile at System, Component and Deployment level (in particular for the latter for what regards HW platform entities)

- «FailureMode»
  - **Extends**
    - UML::Class
  - **Description**
    - This element represents a failure mode of a component. Failure modes should be associated to components’ ports.
  - **Attributes**
    - name : String [1]
    - description : String [0..1]
• exposure : String [0..1]
• controllability : String [0..1]
• likelihood : String [0..1]

«FLABehavior»
  o Extends
    ▪ UML::Component, SysML::Block, UML::Connector,
      UML::InstanceSpecification
  o Description
    ▪ This stereotype should be similar to the one used in CHESS, where
      the failure behavior is described using a FPTC / FI4FA specification.
    ▪ If a <<FlaSpecification>> Connector or InstanceSpecification (as
      instance of Connector) does not have dependability-related
      stereotypes, then it is assumed that failures are propagated “as they
      are”, e.g., one “omission” on one end is propagated as an “omission”
      on the other end. Otherwise dependability annotations can be used to
i) specify a mapping between failure modes on one side, and failure modes on the other side; or ii) specify failure behavior of the connector itself (e.g., a network cable may break). In this case, we should consider (if possible) the failure modes associated to the two ports between which the connector (instance) is placed (through the ConnectorEnd elements in case of the Connector or by using the owned Slots, i.e. the instances of the connected ports, in case of the InstanceSpecification)

- **Attributes**
  - fptc : String [1]
• «ErrorModelBehavior» (replace «DependableComponent»)
  o **Extends**
    - UML::Class, UML::Property, UML::Connector, UML::InstanceSpecification
  o **Description**
    - This stereotype is used to attach an error model to a system element (component/block/connector)
  o **Attributes**
    - errorModel : CHESSML::ErrorModel [1]
• «ErrorModel»
  o Extends
    ▪ UML::StateMachine
  o Description
    ▪ An error model state machine (as in CHESSML)

• «InternalFault»
  o Extends
    ▪ UML::Transition
  o Description
    ▪ An internal fault transition, i.e., a state transition caused by an internal fault.
    ▪ An internal fault can bring the component to a state different from the initial state, but can also make the component move between two internal states.
  o Attributes
    ▪ occurrence : MARTE_Library::NFP_Real [0..1]
    ▪ weight : Double [0..1]
- «InternalPropagation»
  o Extends
    UML::Transition
  o Attributes
    - externalFaults : String [1]
      - Specification of which external faults (i.e., failures of connected components trigger this internal propagation transition). This attribute should be a string complying with the following grammar:

      modeexp ::= Port ‘.’ mode | modeexp ‘AND’ modeexp | modeexp ‘OR’ modeexp | ‘NOT’ modeexp
      mode ::= FailureMode | ‘nofailure’

      As examples, two valid strings are:
      “p1.omission”
      “(p1.commision AND p2.nofailure) OR p3.late”
  o Description
    - An error propagation occurring within the component. The propagation is caused by the occurrence of external faults.
  o Attributes
    - delay : MARTE_Library::NFP_Real [0..1]
    - weight : Double [0..1]
      - the relative probability that this internal propagation occurs with respect to other transitions that may be defined starting from the same state of the error model.

- «Failure»
  o Extends
    UML::Transition
  o Attributes
    - modes : String [1..*]
      - Specification of which failure modes affect which ports of the component. This attribute should be an array of strings complying with the following grammar:

      modes ::= Port ‘.’ mode | Port ‘(’ modeprob ‘)’
      mode ::= FailureMode | ‘nofailure’
      modeprob ::= prob “ : ” FailureMode | modeprob ‘,’ modeprob
      prob ::= Double

      As examples, two valid strings are:
      “p1.omission”
      “p2.(0.2 : omission, 0.8 : commission)”
  o Attributes
    - delay : MARTE_Library::NFP_Real [0..1]
- **weight**: Double [0..1]
  - the relative probability that this internal propagation occurs with respect to other transitions that may be defined starting from the same state of the error model.

- **Description**
  - The occurrence of a failure of the component (i.e., an erroneous component state reaches the service interface). The modes attribute is used to specify which failure modes affect the different ports of the component.
  - Having failures as events/transitions is in line with the “classical” definition given by the taxonomy of Avizienis et al [AV]. There could be for example situations where a component, as a consequence of a failure moves back to the initial state. For example, a component may produce a wrong output as a consequence of a particular input, but then produce correct output in the following.

- **«NormalState»**
  - **Extends**
    - UML::State
  - **Description**
    - A state of the component that is considered “normal”, i.e., in which the component is providing its full functionalities.

- **«ErrorState»**
  - **Extends**
• UML::State
  o Description
    ▪ A state of the component that is considered erroneous (i.e., not complying with the specifications).

• «DegradedState»
  o Extends
    ▪ UML::State
  o Description
    ▪ A state of the component in which it is not delivering its complete functionalities, but which was foreseen in the specifications (e.g., reduced services due to maintenance activities).

• «ErrorDetection»
  o Extends
    ▪ UML::Transition
  o Attributes
    ▪ delay : MARTE_Library::NFP_Real [0..1]
    ▪ successProbability : Double [0..1]
  o Description
    ▪ An error detection transition. Can be used in components’ error models.
  o Constraint
    ▪ from “ErrorState” to any State

• «ErrorHandling»
  o Extends
    ▪ UML::Transition
  o Attributes
    ▪ delay : MARTE_Library::NFP_Real [0..1]
    ▪ successProbability : Double [0..1]
  o Description
    ▪ An error handling transition. Can be used in components’ error models.
- from “ErrorState” to any State

- « FaultHandling »
  - Extends
    - UML::Transition
  - Attributes
    - delay : MARTE_Library::NFP_Real [0..1]
    - successProbability : Double [0..1]
  - Description
    - A fault handling transition. Can be used in components’ error models.
  - Constraint
    - From any State (except the InitialState) to any State
«M&MActivity»
  o Extends
    UML::Activity
  o Attributes
    when : String [1]
    • Specification of when the activity should be triggered. This attribute should be a strings complying with the following grammar:

\[
S ::= T \[ \text{EX} \] | T \[ \text{EX} \] { L }
\]
\[
T ::= \text{Immediately} | \text{AtTime}( \text{R} ) | \text{Periodic}( \text{D} )
\]
\[
\text{EX} ::= ( \text{EX} \text{and} \text{EX} ) | ( \text{EX} \text{or} \text{EX} ) | \text{not} \text{EX} | \text{true} | \text{FD}
\]
\[
\text{FD} ::= \text{Failed}( \text{FailureMode} ) | \text{Failed}( \text{Port}. \text{FailureMode} ) | \text{Detected}( \text{ErrorState} )
\]
\[
L ::= \text{Before}( \text{R} ) | \text{After}( \text{R} ) | \text{Interval}( \text{R} , \text{R} )
\]
\[
R ::= \text{MARTE::NFP_Real}
\]

As examples, two valid strings are:

“Immediately [ p1.omission ]”
“Periodic( 720 ) [ true ] { After(8760) }”

o Description
  • An activity related to maintenance, which may be composed of multiple actions (M&MAction). The “when” attribute specifies the conditions that should hold for this activity to be triggered.

«M&MAction» (abstract)
  o Extends
    UML::Action
  o Attributes
    • duration : MARTE::NFP_Real [0..1]
    • probSuccess : Double [1]
    • targets : UML::Property [*]
o **Description**
   - An action belonging to an M&MActivity.

- «M&MAction»
  o **Extends**
    - UML::Action
  o **Attributes**
    - duration : MARTE_Library::NFP_Real [0..1]
    - probSuccess : Double [1]
  o **Description**
    - An action belonging to an M&MActivity.

- «Repair»
  o **Extends**
    - CHESSML::M&MAction
  o **Attributes**
    - targets : UML::Property [*]
  o **Description**
    - A repair action performed on a component (or set of components). Repair brings the component back to its initial healthy state.

- «Detection»
  o **Extends**
    - CHESSML::M&MAction
  o **Attributes**
    - target : UML::Property [1]
    - detectableStates : CHESSML::ErrorState [1..*]
    - onDetection : CHESSML::M&MAction [*]
  o **Description**
    - An error detection activity performed on a component to detect specific erroneous states defined in its error model.

- «Recovery»
  o **Extends**
    - CHESSML::M&MAction
  o **Attributes**
    - target : UML::Property [1]
    - recoveryState : UML::State [1]
  o **Description**
    - A recovery activity that brings a component in a specific state.

- «StateBasedAnalysis»
  o **Extends**
    - MARTE::GQAM::GaAnalysisContext
  o **Attributes**
    - measure : String [1]
    - measureEvaluationResult : String [0..1]
    - targetInstances : UML::InstanceSpecification [*]
- targetPort : UML::Slot (as instance of Port) [*]
- targetFailureModes : CHESSML::FailureMode [*]
- initialConditions : CHESSML::SBAInitialConditions [*]

  - **Description**
    - Definition of measures of interest for state-based analysis as in CHESS. The target may be specified by specifying i) only the component instances, ii) specific ports of those instances, or iii) specific failure modes of those ports.
    - StateBasedAnalysis can have one or more initial conditions to be applied to the analysis.

- «SBAInitialConditions»
  - **Extends**
    - UML::Component
  - **Attributes**
    - targetInstance : UML::InstanceSpecification [1]
    - setup : String [1]
      - Specification of the probability distribution of initial states of the component. This attribute should be a strings complying with the following grammar:

      SETUP ::= '{' SP '}'
      SP ::= SP ',' SP | S ':' P
      S ::= UML::State | 'Healthy' | F
      F ::= 'Failed(' FailureMode ')' | 'Failed(' Port '.' FailureMode ')
      P ::= MARTE::NFP_Real

      As examples, two valid strings are:

      "{ Failed(p1.omission) : 1.0 }"
      "{ Healthy : 0.8, Erroneous : 0.2 }"

  - **Description**
    - This stereotype is used to specify initial conditions for components of the system (e.g., healthy, failed, etc.).
7.2 NOTES ABOUT THE CURRENT IMPLEMENTATION

All the stereotypes defined in the previous version of the CHESS dependability profile are kept in the current implementation to be able to still use the current transformations and analysis; they appear in this document as deprecated (see below).

7.2.1 Deprecated stereotypes

The following entities (stereotypes and/or properties) from the old CHESS dependability profile are deprecated:

(deprecated → NEW(optional))

- FPTCSpecification.failure → FPTCSpecification.failureMode
- FPTC→ FLABehavior
- FPTCPortSlot→FailureModes
- FailureType→FailureMode
- DependableComponent → ErrorModel
- ErrorState.type
- ErrorState.vanishingTime

7.2.2 Renamed Stereotypes

- Error→ErrorState
- Propagation.propDelay=Propagation.delay

### 7.2.3 About the FPTCSpecification

FPTCSpecification, as defined in the old profile, allows storing failures for ports by using composite structure diagrams, i.e. by using annotated Comments attached to port-part pairs. I.e. the modeller can use the FPTCSpecifications in the composite diagram to provide the faults in input (as fault injection); at the same time the tool uses FPTCSpecifications to back propagate the propagated failures, as resulting from the analysis.

FPTCSpecification is kept in the current version of the profile to allow an initial smooth customization of the implementations currently available for the FPTC analysis; for this goal a dedicated relationship between FailureModes and FPTCSpecification has been added in place of the one between FPTCPortSlot and FPTCSpecification defined in the old profile. So, to reuse the current FPTC implementation, FailureModes should be used in place of the FPTCPortSlot; then FailureModes at Slot level can be linked to the <<FPTCSpecification>> Comments modelled in the composite diagram.

However the final goal (in CONCERTO or after) should be to avoid the use of the FPTCSpecification in place of the usage of the FailureModes stereotype. In other words the user should be able to use the InstanceView to attach failures to ports, in particular by using FailureModes attached to Slots, the latter as instances of ports, without the need to create the FPTCSpecification. So the usage of stereotyped comments, needed to work with the composite diagram for ports decoration, would not be useful anymore.

It is worth noting that the current FPTC analysis already considers the instance level, so it already considers the InstanceSpecifications and Slots entities, retrieving the FPTCSpecification starting from them; the resulting failures are currently back propagated to the FPTCSpecifications just to allow the modeller to see the results by using the composite diagram. So what we should implement is a new view to allow the modeller to check the results directly by looking at the InstanceView, without the need to switch to the composite diagram. The usage of composite diagram also has some limitation in case of hierarchical systems, with a hierarchy level > 2, and the new view for FPTC result built on top of the InstanceView would solve these limitations.
8 CONTRACT PROFILE

The entities of the CHESS Contract profile concerning modelling of contracts are represented in the following UML profile diagram.
The definition and semantic of the aforementioned entities are elaborated further in the next sub-sections.

8.1 CONTRACT

<<Contract>> is a stereotype which extends the SysML ConstraintBlock entity. Contract allows to aggregate two special kind of UML Constraint, in particular it allows the modeling of the assumption and guarantee contract’s properties as Constraint owned by the ConstraintBlock itself.

It is worth noting that the profile does not impose any particular language to be used for the specification of the assume and guarantee contract’s properties.

It is often the case that a contract is specified to put some restriction upon the attributes (i.e. their values) owned by a given component (e.g. component input and output ports): according to the SysML semantics defined for the ConstraintBlock, the attributes which are eventually subject of the assume and guarantee constraints specification (i.e. subject of the contract) can also appear as parameters (i.e. attributes) of the Contract. This allows to model contracts in isolation, so enabling their reuse, while giving the possibility to bind the...
contract and its constrained attributes to a given component and attributes at a later stage in the process. This is analogous to how ConstraintBlocks works in SysML (in particular by using Parametric diagram).

The association between a <<Contract>> and a component is obtained by instantiating a <<ContractProperty>> (see next paragraph) into the component as required by SysML for the ConstraintBlock and the ConstraintProperty entities.

8.2 **CONTRACT PROPERTY AND CONTRACT REFINEMENT**

The <<ContractProperty>> is a stereotype which derives from the SysML ConstraintProperty; it allows to assign Contracts to components. The ContractType attribute allows to model weak or strong contracts assignment.

The <<ContractProperty>> stereotype has a RefinedBy attribute that refers the set of ContractProperties that decompose it. The usage of the ContractRefinement data type allows the modelling of contracts decomposition at the level of the contracts instantiations, i.e. at the level of ContractProperties defined for the parent and child components (the latter modelled as parts, i.e. Property, in a block definition diagram).

![Diagram of Contracts](image)

**Figure 19 Contracts**

In the figure above the association of Contracts to Blocks is highlighted in bright green: e.g. the System_Brake_Time Contract is instantiated in the System Block by means of the ConstraintProperty brake_time typed as System_Brake_Time.
The Formal Properties that Contracts use as their Assume and Guarantee are highlighted in dark red: e.g. System_Brake_Time uses the Formal Property named System_Brake_Time_Assumption as its Assume and the Formal Property named System_Brake_Time_Guarantee as its Guarantee.

8.3 COMPONENT\BLOCK INSTANCES AND WEAK CONTRACTS

The ComponentInstance stereotype allows to provide the list of weak contracts which actually hold for a given component instance.

8.4 ABOUT MAPPING TO OCRA

Regarding mapping to OCRA: an implicit mapping is performed in the NuSMV3 OCRA analysis by comparing the string of the Assume/ Guarantee parameters with the name of the properties (e.g. ports) of the Block where the Contract has been instantiated. In the example, for instance, the parameters used in the System_Brake_Time contract’s Assume and Guarantee are No_Double_Fault, Pedal_Pos1, Pedal_Pos2 and Brake_Line; an implicit binding is assumed between such parameters and the properties in the System Block named No_Double_Fault, Pedal_Pos1, Pedal_Pos2 and Brake_Line. Identical color underlining is used in the figure to identify such kind of implicit binding. … (to be completed)

9 REQUIREMENT

In CHESS the entities available in the SysML::Requirement package are imported.

9.1 ENTITIES

9.2 REQUIREMENT

From SysML::Requirements

9.3 DERIVED REQT

From SysML::Requirements

9.4 SATISFY

From SysML::Requirements

References

10 REFERENCES
