CHESS

CHESS Modelling Language

A UML/MARTE/SysML profile

Prepared by Intecs

1 TABLE OF CONTENTS

2 LIST OF FIGURES

3 DOCUMENT HISTORY

Date	Eclipse release	Changes
15 April 2020	1.0.0	Update with AMASS results

4 INTRODUCTION

This document addresses the specification of the CHESS Modelling Language (CHESSML) which objective is to support the model-based specification, design and verification of cyber physical systems as foreseen by the CHESS methodology [CHESS].

CHESSML is defined as a *collection-extension of subsets of standard OMG languages* (UML2.5, MARTE 1.2, SysML1.2); in particular sub-profiles supporting contract-based and dependability concerns have been defined, while MARTE has been adopted (with minor deviations) for what concern the timing perspective.

Figure below gives a conceptual view about the CHESSML dependencies.

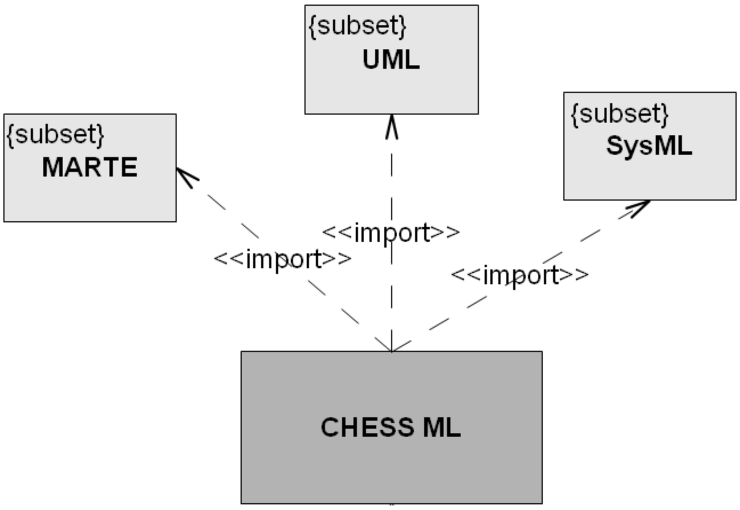


Figure 1: CHESSML dependencies

In the following sections each part of the profile is explained.

5 CORE PROFILE

This part is related to the customization of the UML Model

5.1 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.

6 VIEWS PROFILE

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

6.1 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 7 for further restriction upon these entities).

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

- For predictability: CHRtSpecification (see section 8)
- For dependability: the entities described in section 9.

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 6.

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.

7 Component Model Profile

The component model package defines a set of concepts that can be used to model CHESS software component model.

7.1 Entities

7.1.1 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*".

7.1.2 Realization

From UML.

Additional Constraints:
From ComponentImplementation to ComponentType.

7.1.3 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 (trough *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.

7.1.4 FlowPort

From MARTE::GCM.

7.1.5 Property

From UML.

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

7.1.6 Operation

From UML.

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

Additional Constraints:

[1] *Operation::preCondition.* Excluded from the CHESS ML.

[2]*Operation::postCondition.* Excluded from the CHESS ML.

[3]*Operation::bodyCondition.* Excluded from the CHESS ML.

7.1.7 Interface

From UML.

7.1.8 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.

7.1.9 DataType

From UML

7.1.10 InterfaceRealization

From UML.

Not mandatory in CHESS.

7.1.11 Dependency

From UML.

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

7.1.12 Enumeration

From UML.

7.1.13 EnumerationLiteral

From UML.

7.1.14 InstanceSpecification

From UML.

7.1.15 Slot

FromUML

7.1.16 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.

7.1.17 ModeBehavior

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

Note: not currently supported by model transformations.

7.1.18 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.

7.1.19 ModeTransition

From MARTE::CoreElements.

Note: not currently supported by model transformations.

7.1.20 Configuration

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

7.1.21 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

7.1.22 Interaction

From UML.

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

Under evaluation in CHESS.

7.1.23 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.

7.2 STEREOTYPES

7.2.1 ComponentType

Maps the component type notion of the CHESS component model.

Extension:
Component

Attributes

None

Associations:
None

Constraints:
[1] It cannot own behaviours.

7.2.2 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 be completed

8 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 2) 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.

[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

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\visualzed through the UML composite structure diagrams.

The MARTE HLAM::RtFeature is extended in CHESSML (as CHRtPortSlot stereotype, see Figure 2) to allow application of CHRtSpecification to a given instance of component’s port (Slot in UML).

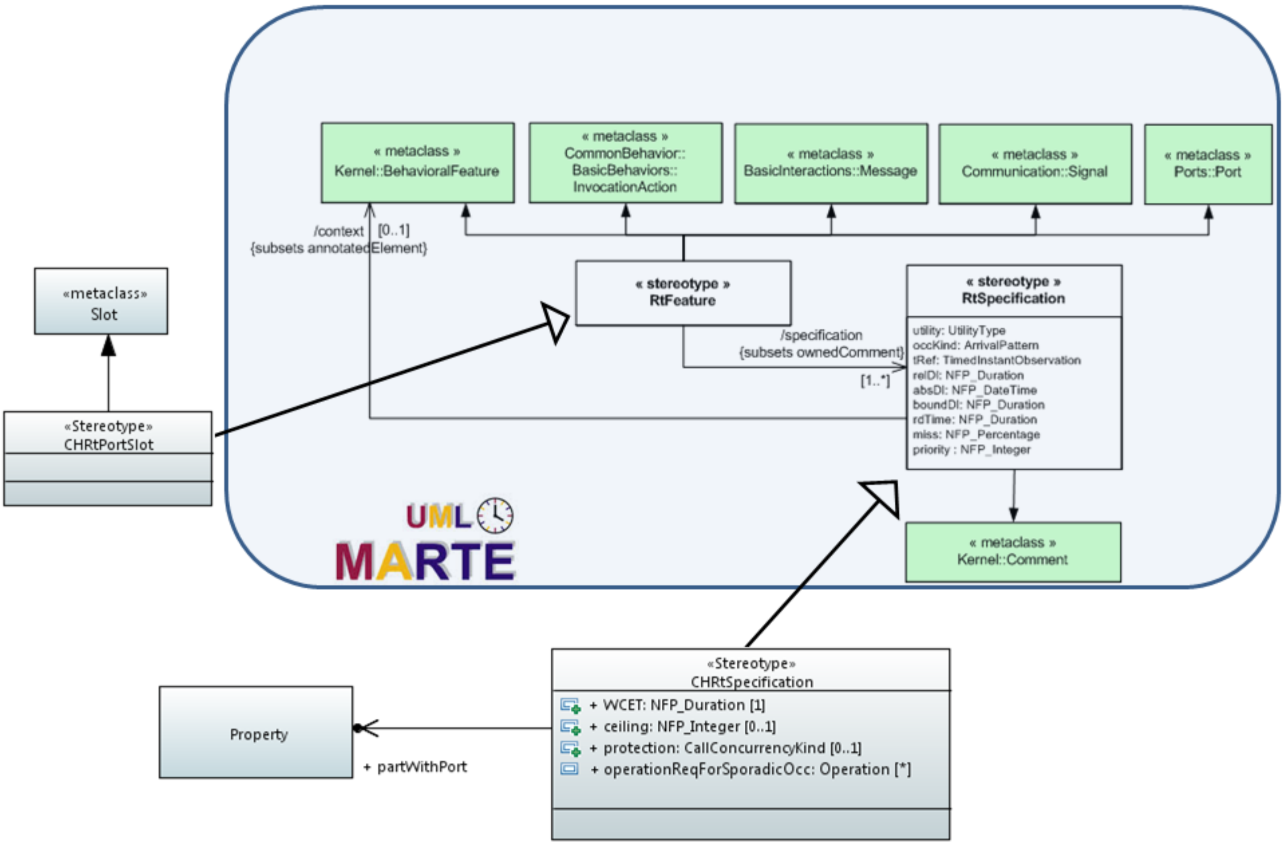


Figure 2: MARTE HLAM::RtSpecification and HLAM::RTFeature CHESSML extensions

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 SRM[3]::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)

MARTE Alloc[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[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) .

MARTE GaResourcePlatform

CHESS Extends the MARTE::GQAM::GaResourcePlatform with CHGaResourcePlatform, to make it applicable 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

Analysis context (CHESS AnalysisView)

CHESSML uses the MARTE SAM[6]::SaAnalysisContext entity to specify the information needed to perform a given analysis.

[3] MARTE Software Resource Modeling (SRM) sub-profile
[4] MARTE Allocation Modeling sub-profile
[5] MARTE Non-functional Properties Modeling (NFPs) sub-profile

MARTE GQAM[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::GaWorkloadBeahavior and SAM::SaEndtoEndFlow are used in CHESSML (in the AnalysisView) to refer a sequence diagram modelled in the ComponentView (in detail, GaWorkloadBeahavior 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.

MARTE SRM[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.

[6] Schedulability Analysis Modeling (SAM) sub-profile
[7] Generic Quantitative Analysis Modeling (GQAM) sub-profile
[8] Software Resource Modeling (SRM) sub-profile

- 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.

9 DEPENDABILITY PROFILE

This section describes the CHESS modelling language for dependability. The profile is based upon the SafeConcert conceptual dependability model [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 3.

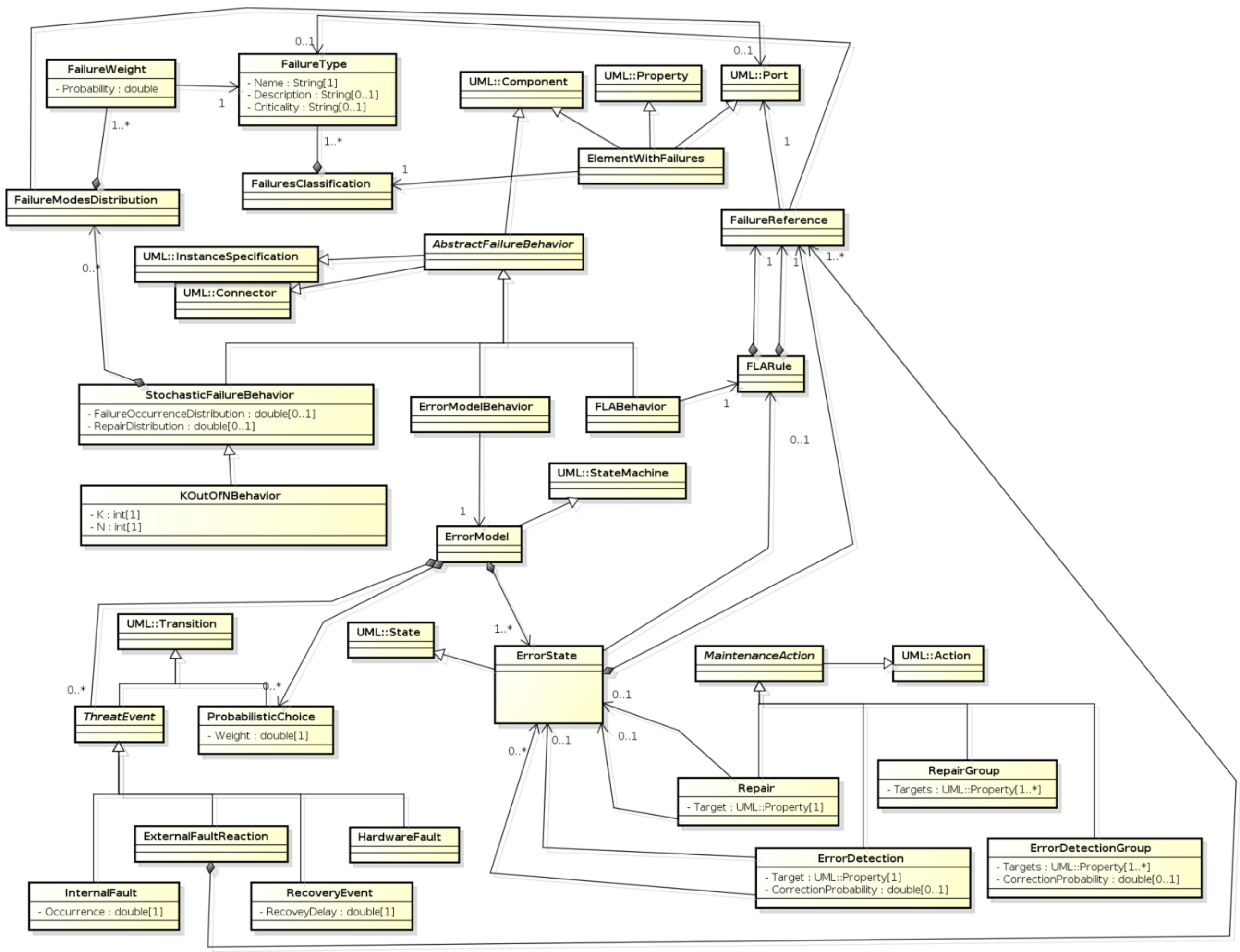
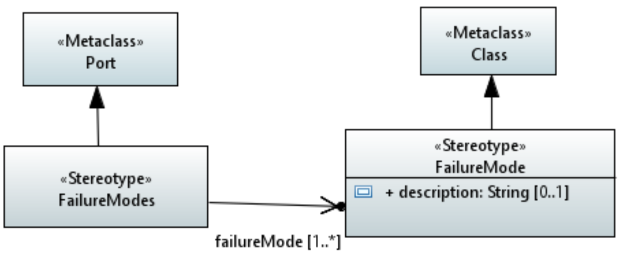


Figure 3: Proposal for the dependability conceptual model.

The UML profile implementing the aforementioned conceptual model is presented in the following sub-sections. Figure below provides a summary of the stereotypes addressing error behaviour modelling.



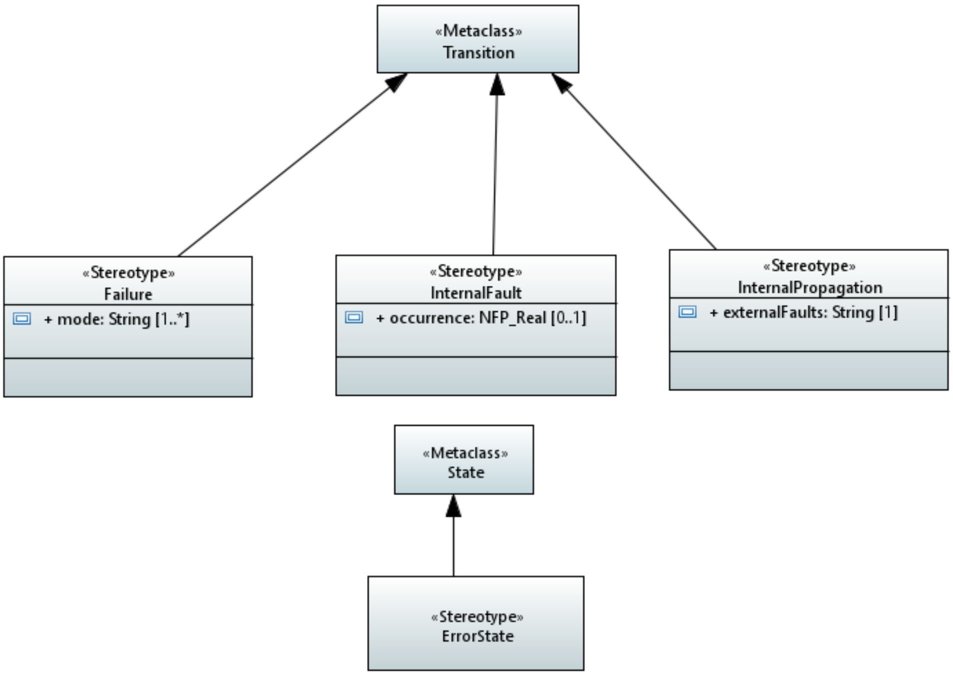
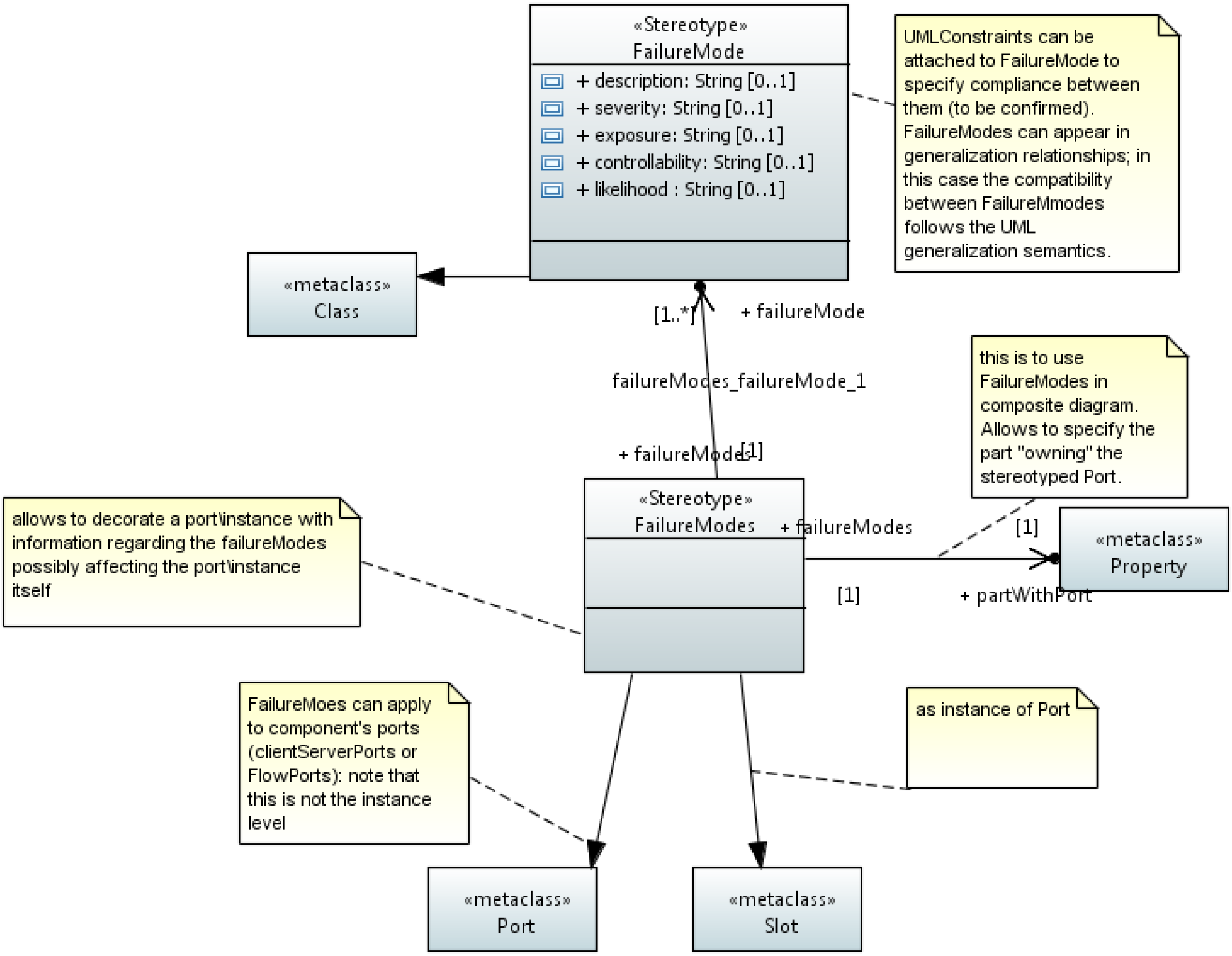


Figure 4. CHESS dependability profile excerpt for error behaviour modelling

9.1 Stereotypes - Safety

Here we list the dependability profile stereotypes defined as implementation of the conceptual model presented in
The profile is defined here by taken 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)

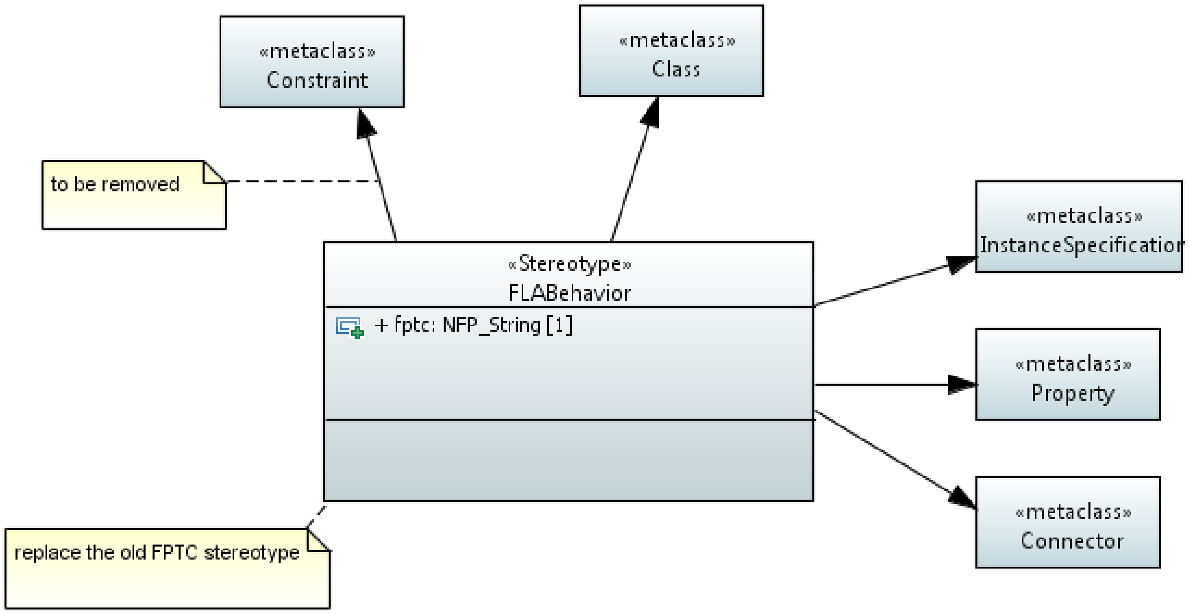
- «FailureMode»
  - *Extends*
    - UML::Class
  - *Description*
    - This element represents a failure mode of a component. Failure modes should be associated to components' ports.
  - *Attributes*
    - name : String [1]
    - description : String [0..1]
    - exposure : String [0..1]
    - controllability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



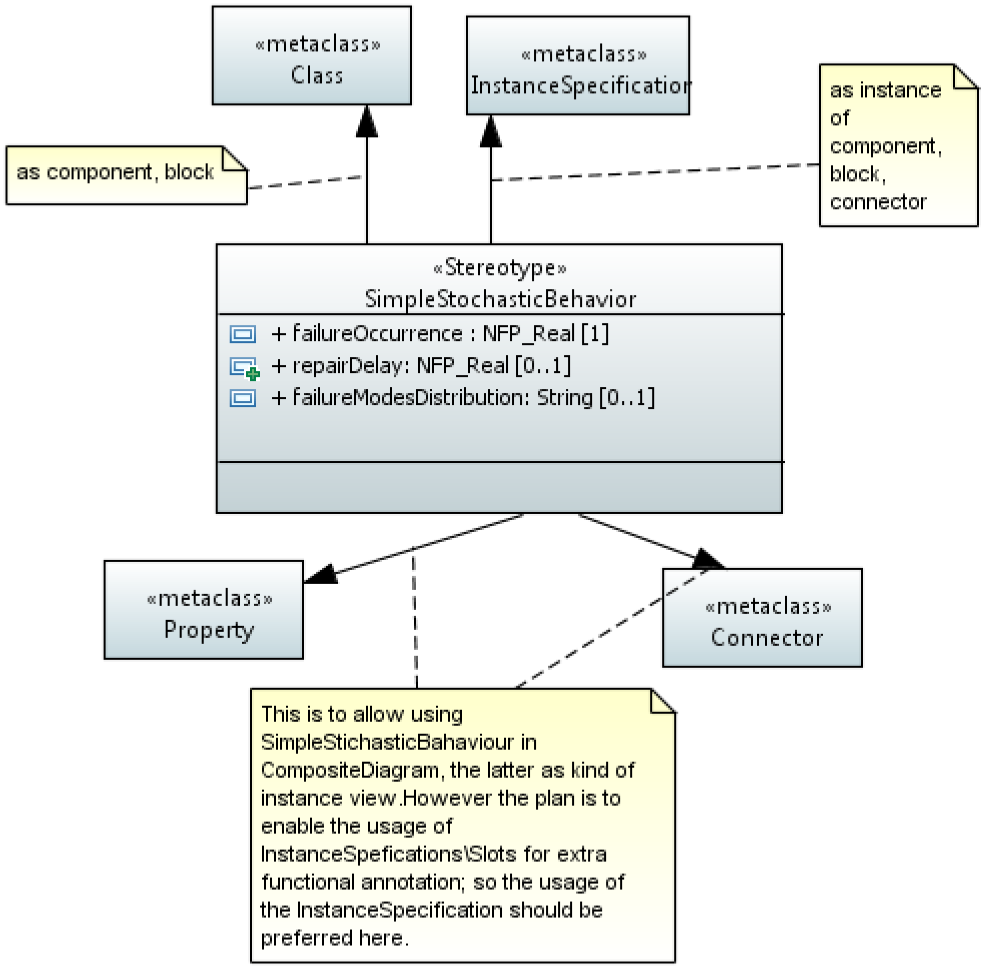
- «FLABehavior»
  - *Extends*
    - UML::Component, SysML::Block, UML::Connector, UML::InstanceSpecification
  - *Description*
    - This stereotype should be similar to the one used in CHESS, where the failure behavior is described using a FPTC / FI4FA specification.
    - If a <<FlaSpecification>> 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)

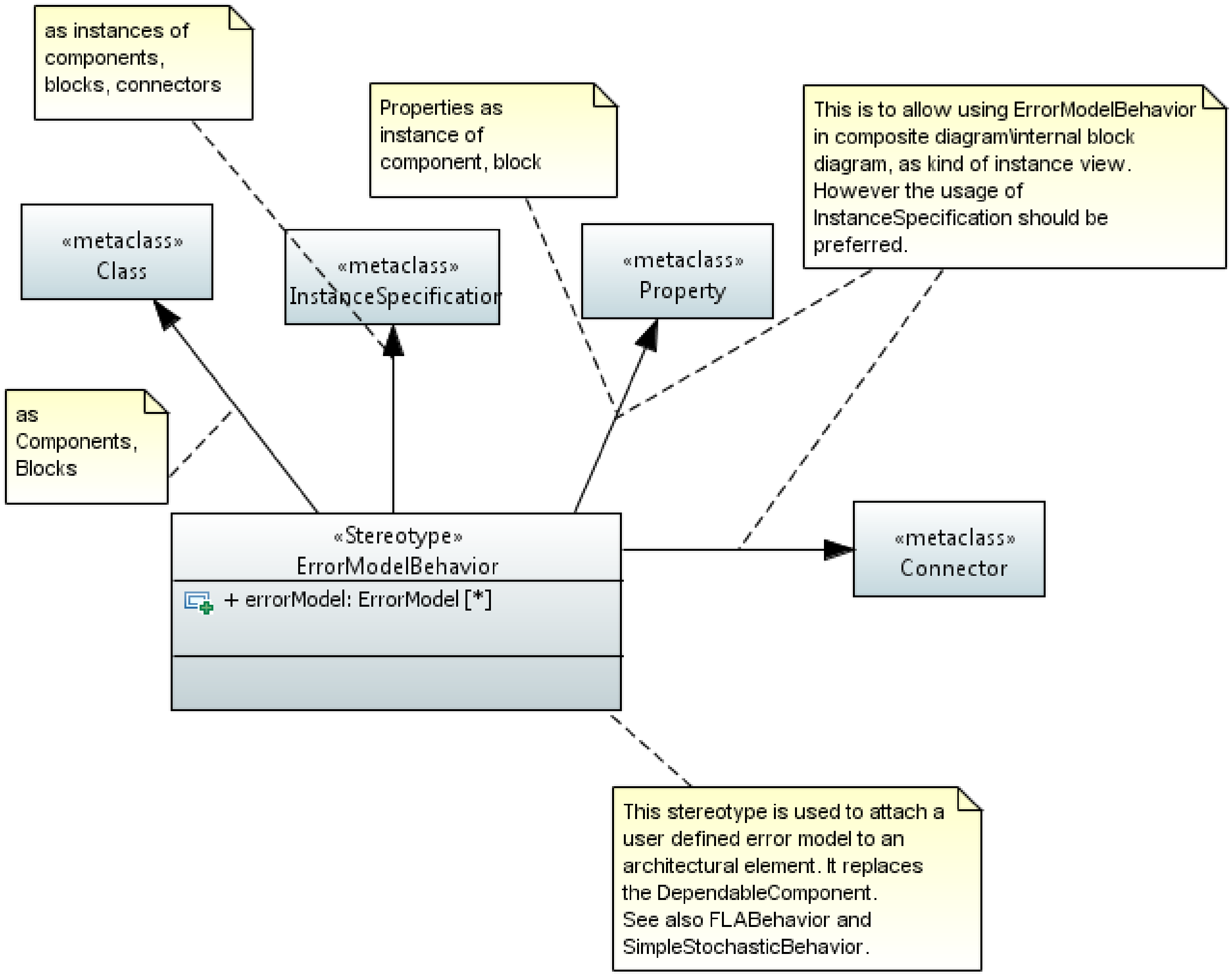
- *Attributes*
  - fptc : String [1]



- «SimpleStochasticBehavior»
  - *Extends*
    - UML::Class, UML::Property, UML::Connector, UML::InstanceSpecification
  - *Description*
    - This stereotype is meant to replace the Stateless/Stateful Software/Hardware stereotypes (those were considered difficult to handle/understand by industrial users in CHESS).
  - *Attributes*
    - failureOccurrence : MARTE_Library::NFP_Real [1]
    - repairDelay : MARTE_Library::NFP_Real [0..1]
    - failureModesDistribution : String [0..1]



- «ErrorModelBehavior» (replace «DependableComponent»)
    - *Extends*
        - UML::Class, UML::Property, UML::Connector, UML::InstanceSpecification
    - *Description*
        - This stereotype is used to attach an error model to a system element (component/block/connector)
    - *Attributes*
        - errorModel : CHESSML::ErrorModel [1]



- «ErrorModel»
  - *Extends*
    - UML::StateMachine
  - *Description*
    - An error model state machine (as in CHESSML)

- «InternalFault»
  - *Extends*
    - UML::Transition
  - *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.
  - *Attributes*
    - occurrence : MARTE_Library::NFP_Real [0..1]
    - weight : Double [0..1]

- «InternalPropagation»
  - *Extends*
    - UML::Transition
  - *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"
  - *Description*
    - An error propagation occurring within the component. The propagation is caused by the occurrence of external faults.
  - *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.

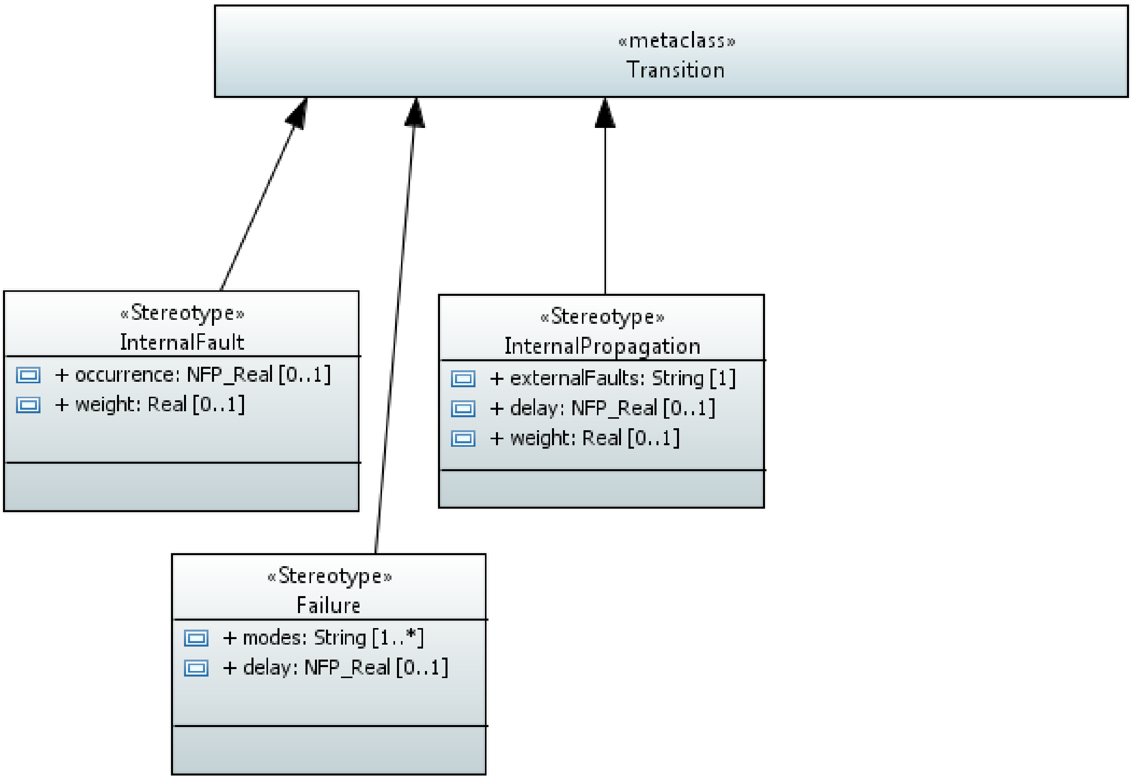
- «Failure»
  - *Extends*
    - UML::Transition
  - *Attributes*
    - modes : String [1..*]
      - Specification of which failure modes affect which ports of the component. This attribute should be an array of strings complying with the following grammar:

      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)"
    - 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.

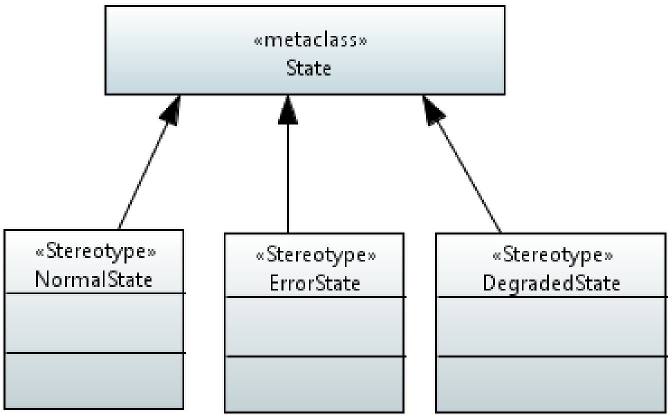


- «NormalState»
    - *Extends*
        - UML::State
    - *Description*
        - A state of the component that is considered "normal", i.e., in which the component is providing its full functionalities.

- «ErrorState»
    - *Extends*

        - UML::State
    - *Description*
        - A state of the component that is considered erroneous (i.e., not complying with the specifications).

- «DegradedState»
    - *Extends*
        - UML::State
    - *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).

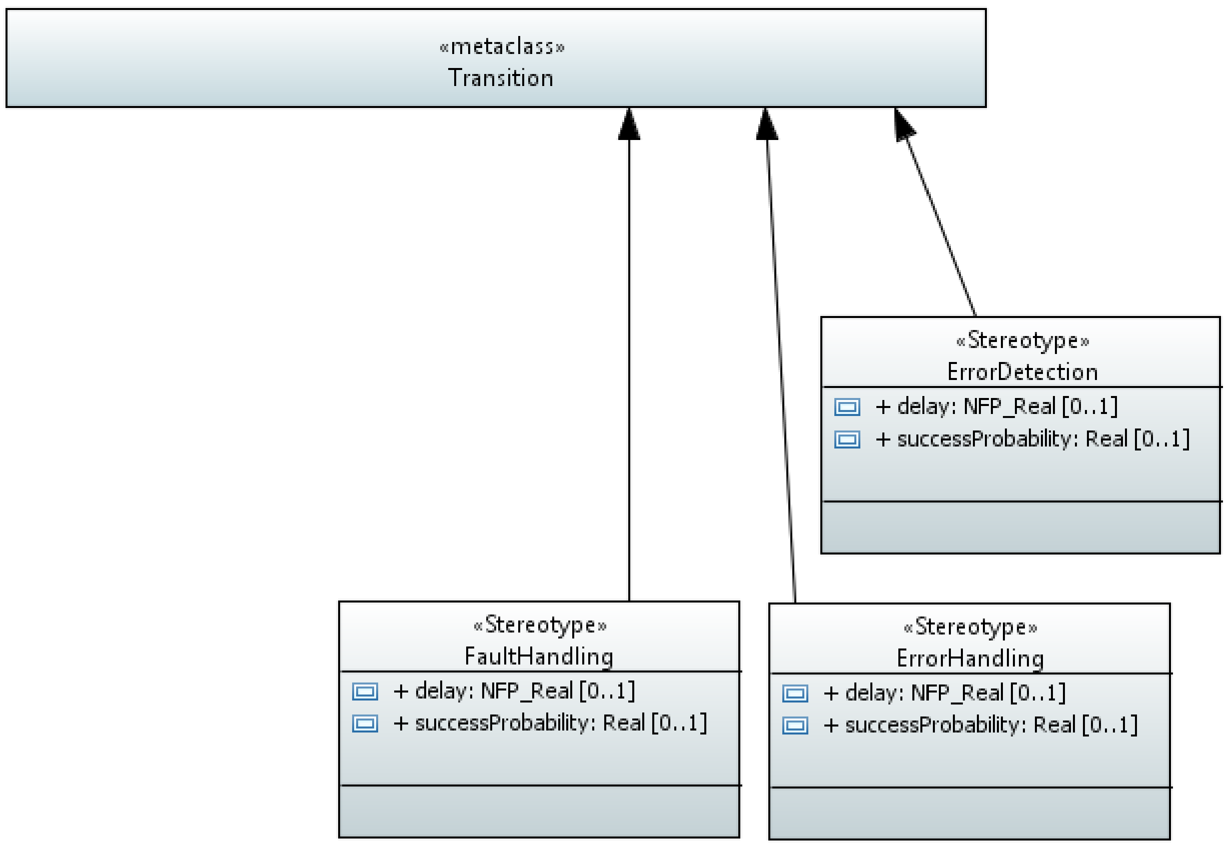


- «ErrorDetection»
    - *Extends*
        - UML::Transition
    - *Attributes*
        - delay : MARTE_Library::NFP_Real [0..1]
        - successProbability : Double [0..1]
    - *Description*
        - An error detection transition. Can be used in components' error models.
    - *Constraint*
        - from "ErrorState" to any State

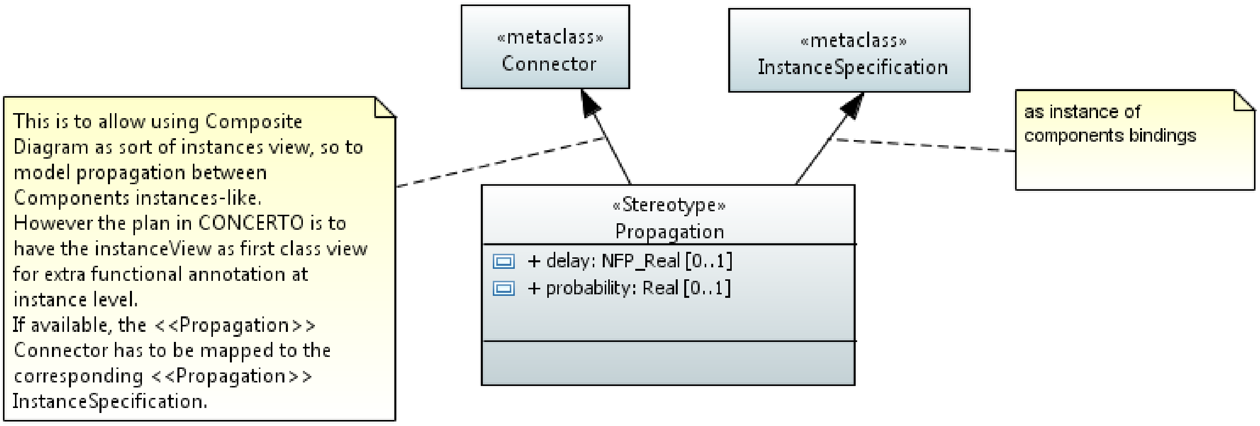
- «ErrorHandling»
    - *Extends*
        - UML::Transition
    - *Attributes*
        - delay : MARTE_Library::NFP_Real [0..1]
        - successProbability : Double [0..1]
    - *Description*
        - An error handling transition. Can be used in components' error models.
    - *Constraint*

    - 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



- «Propagation»
  - *Extends*
    - UML::Connector, UML::InstanceSpecification
  - *Attributes*
    - delay : MARTE_Library::NFP_Real [0..1]
    - probability : Double [0..1]



- «M&MActivity»
  - *Extends*
    - UML::Activity
  - *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 '[' EX ']' | T '[' EX '] {' L '}'
      T ::= 'Immediately' | 'AtTime(' R ')' | 'Periodic(' D ')'
      EX ::= '(' EX 'and' EX ')' | '(' EX 'or' EX ')' | 'not' EX | 'true' | FD
      FD ::= 'Failed(' FailureMode ')' | 'Failed(' Port '.' FailureMode ')' |
      'Detected(' ErrorState ')'
      L := 'Before(' R ')' | 'After(' R ')' | 'Interval(' R ',' R ')'
      R ::= MARTE::NFP_Real

      As examples, two valid strings are:

      "Immediately [p1.omission]"
      "Periodic(720) [true] { After(8760) }"
  - *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*)
  - *Extends*
    - UML::Action
  - *Attributes*
    - duration : MARTE_Library::NFP_Real [0..1]
    - probSuccess : Double [1]
    - targets : UML::Property [*]

- *Description*
    - An action belonging to an M&MActivity.

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

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

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

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

- «StateBasedAnalysis»
  - *Extends*
    - MARTE::GQAM::GaAnalysisContext
  - *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.

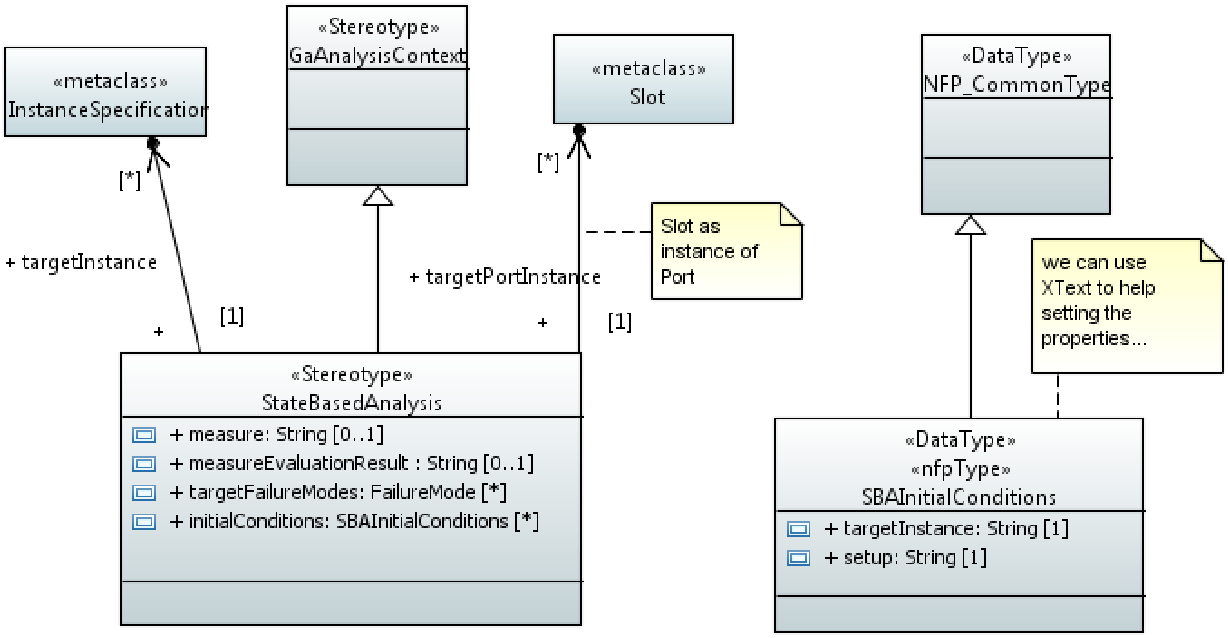
- «SBAInitialConditions»
    - *Extends*
        - UML::Component
    - *Attributes*
        - targetInstance : UML::InstanceSpecification [1]
        - setup : String [1]
            - Specification of the probability distribution of initial states of the component. This attribute should be a strings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



- <u>Other comments</u>
  - «FLABehavior» and «SimpleStochasticBehavior» could in principle coexist on the same component.
- «SANAnalysis»
  - *Extends*
    - MARTE::GQAM::GaAnalysisContext
  - *Attributes*
    - None
  - *Description*
    - Used to for CHESS to SAN analysis

9.2 STEREOTYPES - SECURITY

The CHESS dependability profile includes stereotypes related to security concerns, as showed in Figure 5.

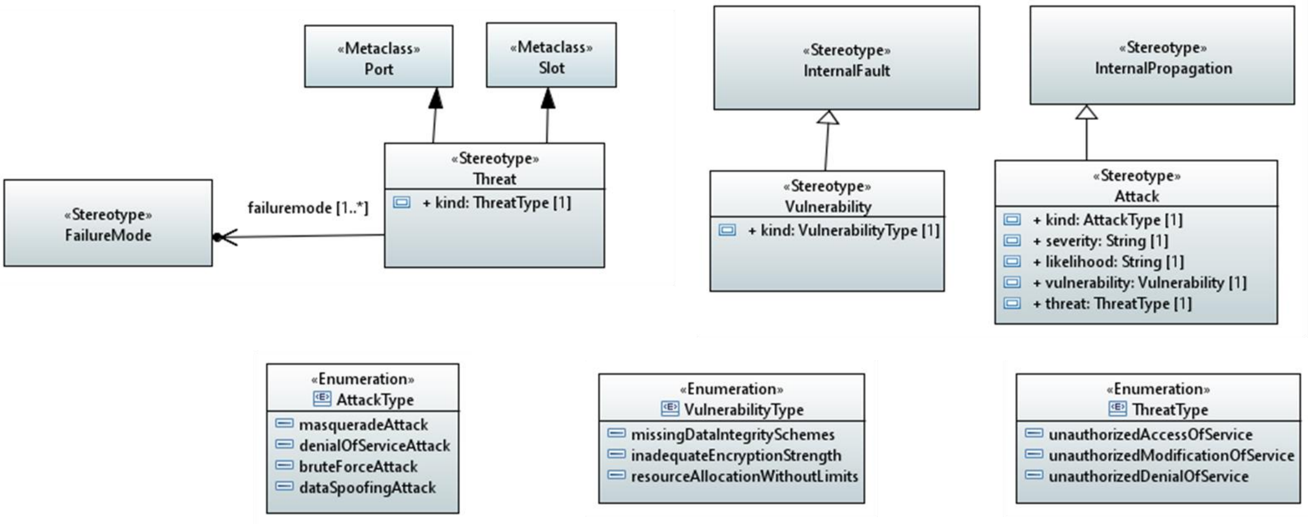


Figure 5. Security Profile

9.2.1 Attack

<u>Extends Metaclass</u>

UML Message

<u>Superclass</u>

InternalPropagation

<u>Attributes</u>

- Kind: *AttackType* [1]
  The type of the attack
- Severity: *String* [1]
  The severity of the attack
- Likelihood: *String* [1]
  The likelihood of the attack
- Threat: *Threat* [1]
  The security breach caused by the attack

<u>Relationships</u>

- Vulnerability: Vulnerability [1]
  The vulnerability exploited by the attack.

<u>Semantics</u>

Attack represents an attempt to expose, alter, disable, destroy, steal or gain unauthorized access to or make unauthorized use of an asset. An attack rises due to the associated threat and cause the actual breach, if able to exploit a Vulnerability. The attack is a specialized InternalPropagation referring to the erroneous transition due to an external fault.

9.2.2 Vulnerability

<u>Extends Metaclass</u>

UML::Port

<u>Superclass</u>

InternalFault

<u>Attributes</u>

- Kind: *VulnerabilityType* [1]
  The type of the vulnerability

<u>Relationships</u>

None

Semantics

Vulnerability refers to an internal weakness of a system.
Vulnerability can be associated to a component port, through which an attack can be attempted.

9.2.3 Threat

Extends Metaclass

UML Port, UML Slot

Superclass

None

Attributes

- Kind: *ThreatType* [1]
  The type of the threat

Relationships

None

Semantics

Threat is an event or situation that can potentially breach the security and cause harm to an asset; it affects a component port (or port instance, represented by the Slot UML metaclass).

9.2.4 Adversary

Extends Metaclass

UML Actor

Superclass

None

Attributes

None

Relationships

None

Semantics

Represent an adversary actor leading to attacks.

9.2.5 AttackScenario

Extends Metaclass

UML Interaction

Superclass

None

Attributes

- Frequency : NFPReal [0..1]
- probSuccess : NFPReal [0..1]

Relationships

None

Semantics

Represent an Interaction invoving an adversary and the system component under attacks. Attacks are modelled with Attack messages exploiting component vulnerabilities.

9.2.6 AttackType

AttackType is an enumeration type that defines literals used for specifying the kind of security attacks.

Literal values are:

- masqueradeAttack
- denialOfServiceAttack
- bruteForceAttack
- dataSpoofingAttack

9.2.7 VulnerabilityType

VulnerabilityType is an enumeration type that defines literals used for specifying the kind of vulnerabilities.
Literal values are:

- missingDataIntegritySchemes
- inadequateEncryptionStrenght
- resourceAllocationWithoutLimits

9.2.8 ThreatType

ThreatType is an enumeration type that defines literals used for specifying the kind of security threats.
Literal values are:

- unauthorizedAccessOfService
- unauthorizedModificationOfService
- unauthorizedDenialOfService

9.3 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).

9.3.1 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

9.3.2 Renamed Stereotypes

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

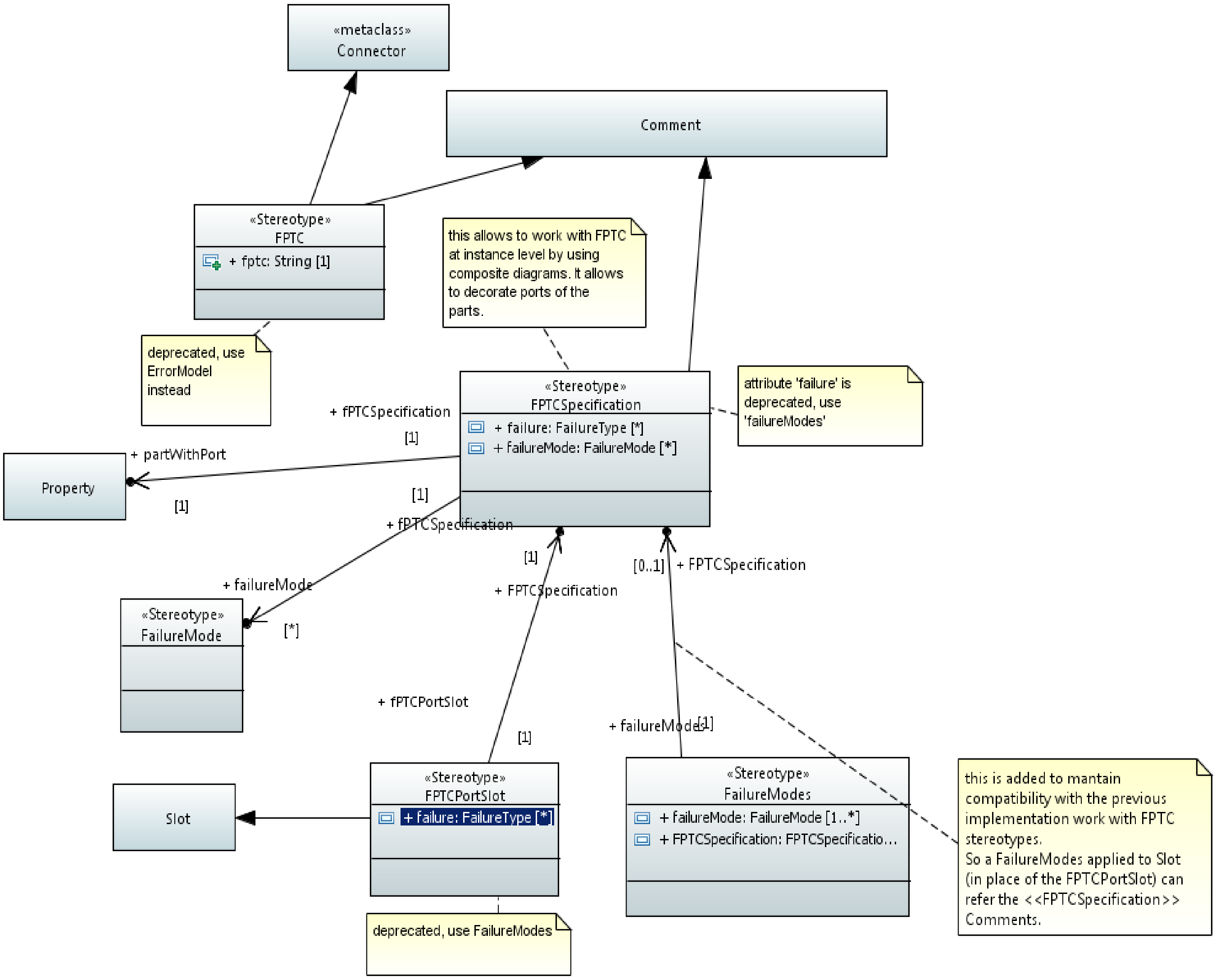
9.3.3 About the FPTCSpecification

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.


«metaclass»
Connector
Comment
«Stereotype»
FPTC
+ fptc: String [1]
deprecated, use ErrorModel instead
this allows to work with FPTC at instance level by using composite diagrams. It allows to decorate ports of the parts.
«Stereotype»
FPTCSpecification
+ failure: FailureType [*]
+ failureMode: FailureMode [*]
attribute 'failure' is deprecated, use 'failureModes'
+ fPTCSpecification
[1]
+ partWithPort
Property
[1]
[1]
+ fPTCSpecification
+ failureMode
«Stereotype»
FailureMode
[*]
[1]
+ FPTCSpecification
[0..1]
+ FPTCSpecification
+ fPTCPortSlot
[1]
+ failureModes
[1]
Slot
«Stereotype»
FPTCPortSlot
+ failure: FailureType [*]
deprecated, use FailureModes
«Stereotype»
FailureModes
+ failureMode: FailureMode [1..*]
+ FPTCSpecification: FPTCSpecificatio...
this is added to mantain compatibility with the previous implementation work with FPTC stereotypes.
So a FailureModes applied to Slot (in place of the FPTCPortSlot) can refer the <<FPTCSpecification>> Comments.

10 CONTRACT PROFILE

The entities of the CHESS Contract profile concerning modelling of contracts are represented in the following UML profile diagram.

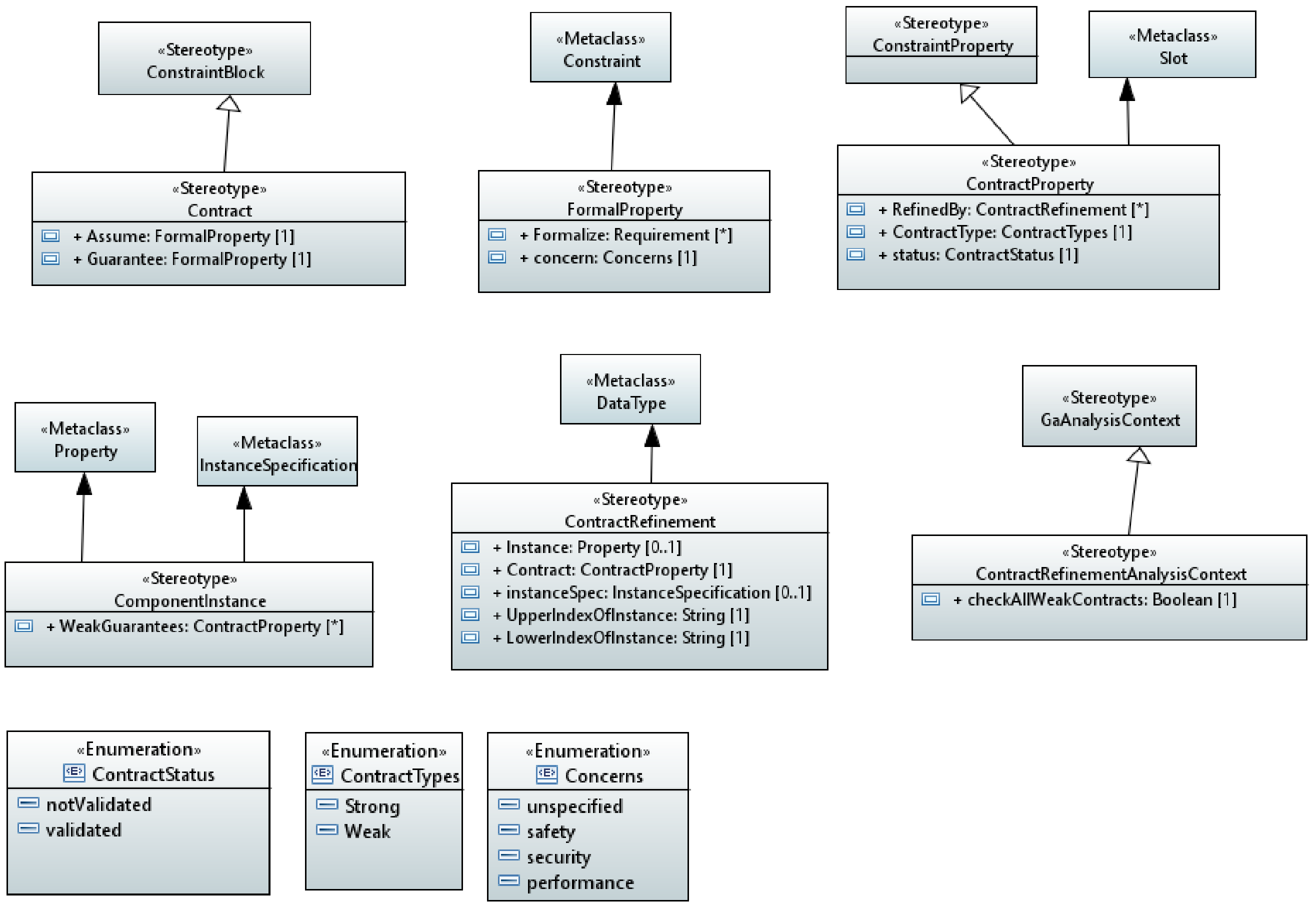


Figure 6. CHESS Contract Profile

The definition and semantic of the aforementioned entities are elaborated in the next sub-sections.

10.1 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 modelling 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 10.3) into the component as required by SysML for the ConstraintBlock and the ConstraintProperty entities.

Extends Metaclass

None

Superclass

- SysML ConstraintBlock

Attributes

None

Relationships

- Assume : *FormalProperty* [1]
  The contract assumption
- Guarantee: *FormalProperty* [1]
  The contract guarantee

Semantics

A Contract models a contract as a pair of assumption and guarantee properties.

10.2 FormalProperty

FormalProperty extends the Constraint element of UML and is used to express restrictions about the possible behaviour of a component or to express the environment behaviour expected by a component; this information is typically derived starting from the available system or component requirements. The specification of a FormalProperty can be provided by using formal languages and by providing expressions upon the available component properties, e.g. its input and output ports. A FormalProperty can then appear as assumption or guarantee property of a Contract.

Extends Metaclass

None

Superclass

UML Constraint

Attributes

None

Relationships

- formalize: SysML *Requirements* [*]
  The requirements formalized by the formal property.
- concern: *Concern* [1]
  The concern addressed by the formal property.

Semantics

A FormalProperty allows to represent the concept of assumption or guarantee property.

10.3 ContractProperty

The ContractProperty stereotype derives from the SysML ConstraintProperty. A ContractProperty allows to bind a Contract to a component, as weak or strong contracts.

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).

Extends Metaclass

None

Superclass

- SysML *ConstraintProperty*

Attributes

None

Relationships

- RefinedBy: *ContractRefinement* [*]
  The properties through which the contract refinement is modelled.
- ContractType: *ContractTypes* [1]
  The kind to be considered for the contract, i.e. weak or strong.

Semantics

ContractProperty allows to bind a Contract to a given component; in particular, a component that has a property stereotyped as ContractProperty and typed with a Contract has that Contract associated. It also allows to specify if the given Contract is bound as weak or strong contract.

10.4 CONTRACTREFINEMENT

ContractRefinement is a data type used to aggregate a Property typed with a component and a ContractProperty

Extends Metaclass

UML DataType

Superclass

None

Attributes

- UpperIndexOfInstance: *String* [1]
- If the owner/s of the refining contract (that is specified in the 'Instance' relationship) is/are part of an array of components, this attribute is the upper index of the range that defines the selected owner/s in the array. LowerIndexOfInstance: String [1]
  If the owner/s of the refining contract (that is specified in the 'Instance' relationship) is/are part of an array of components, this attribute is the lower index of the range that defines the selected owner/s in the array.

Relationships

- Contract: *ContractProperty*
  The refining Contract
- Instance: UML *Property* [0..1]
  The instance (represented in the model as UML Property) owning the refining contract
- instanceSpec: UML *InstanceSpecification* [0..1]
  The instance (represented in the model as UML InstanceSpecification) owning the refining contract

Semantics

ContractRefinement represent a contract associated to a given component instance; this information is then used to model the contract refinement (see ContractProperty).

10.5 COMPONENTINSTANCE

The ComponentInstance stereotype allows to provide the list of weak contracts which actually hold for a given component instance (see also section **Error! Reference source not found.** about odelling of component instances).

Extends Metaclass

UML Property, UML InstanceSpecification

Superclass

None

Attributes

None

Relationships

- weakGuarantee: *ContractProperty* [*]
  The weak contracts which hold for the current component instance.

Semantics
ComponentInstance allows to specify the set of weak contracts that are valid for the given component instance.

10.6 ContractRefinementAnalysisContext

ContractRefinementAnalysisContext allows to collect the proper information to enable contract refinement analysis.

Extends Metaclass

- MARTE GaAnalysisContext

Superclass
None

Attributes

- CheckAllWeakContract: *Boolean* [1]

If True allows to consider weak contracts validation

Relationships
None

Semantics
Aggregates information available to enable contract refinement analysis.

10.7 ContractType

ContractTypes is an enumeration type that defines literals used for specifying if a given contract is a weak or strong one.

Literal values are:

- Strong: indicates a strong contract
- Weak: indicates a weak contract

10.8 ContractStatus

ContractStatus is an enumeration type that defines literals used for specifying the validation status of a given contract.

Literal values are:

- notValidated: indicates a contract that has not been validated
- validated: indicates a contract that has been validated

10.9 Concern

Concern is an enumeration type that defines literals used for specifying the concern of a given formal properties.

Literal values are:

- Unspecified
- Safety
- Security
- Performance

11 Instantiate the parameterized architecture

The CHESS modelling language supports the modelling of parameterized architectures and their instantiation. This section describes this part.

11.1 InstantiatedArchitectureConfiguration

The InstantiatedArchitectureConfiguration is a stereotype which extends the Property element of UML.

Extends Metaclass

None

Superclass

- UML Property

Attributes

- ParameterList: *String* [*]
  The list of pair <ParameterName, ParameterValue>

Relationships

- InstantiatedRootComponent: *Class* [0..1]
  The optional reference to the System Block of the instantiated architecture.

12 ARCHITECTURAL PATTERNS

CHESSML includes an UML profile for patterns specification and instantiation in a given system model; this section provides the description of the stereotypes part of this profile.

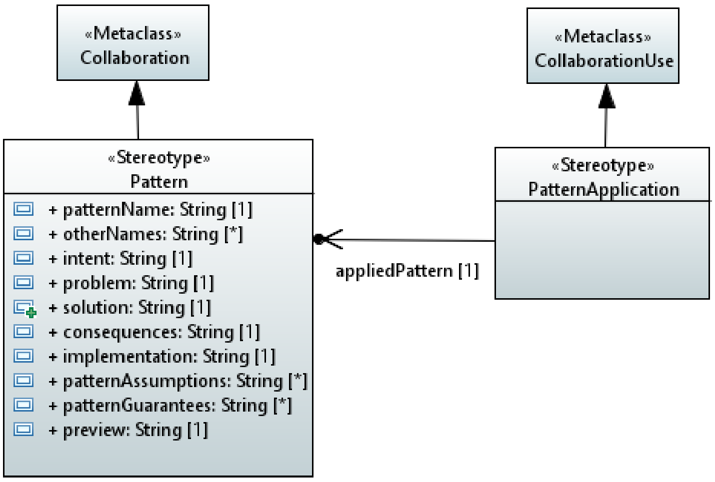


Figure 7. Pattern profile

12.1 PATTERN

Extends Metaclass

UML Collaboration

Superclass

None

Attributes

- patternName: *String* [1]
  The name of the pattern.
- otherNames: *String* [*]
  Other names with which the design pattern can be known in different domains of application or standards.
- Intent: *String* [1]
  The context where the pattern is used. For example, define if the pattern is recommended for a specific safety-critical domain.
- Problem: *String* [1]
  The description of the problem addressed by the design pattern.
- Solution: *String* [1]
  The solution to the problem under consideration. Main elements of the patterns are described.

- Consequences: *String* [1]
  The implication or consequences of using this pattern. It explains both the positive and negative implications of using the design pattern.
- Implementation: *String* [1]
  The set of points that should be taken under consideration when implementing the pattern. Language dependent.
- PatternAssumption: *String* [*]
  The contract assumptions related to the design pattern.
- PatternGuarantee: *String* [*]
  The contract guarantee related to the design pattern.
- Preview: *String* [1]
  The reference to an image showing the pattern with the collaborating entities.

Relationships

None

Semantics

Pattern represents an architectural design pattern to help designer and system architect when choosing suitable solutions for commonly recurring design problem related to functional and dependability (performance, safety, security) concerns. It extends the UML Collaboration entity.

12.2 PatternApplication

Extends Metaclass

UML CollaborationUse

Superclass

None

Attributes

None

Relationships

- appliedPattern: *Pattern* [1]

The pattern that has been applied.

Semantics

PatternApplication represent an instantiation of a given pattern in the context of an actual system.

13 Domain Specific Support: Avionics

13.1 Introduction

CHESSML provides support for modelling Integrated Modular Avionics (IMA) principles[9].

An IMA partition is an aggregate of component instances that represent ARINC-653 processes, which may also include non-ARINC-process components that provide utility operations

Precedence and exclusion relations of operations/functions are expressed as property attributes of function interfaces, to determine the operations that can run together (and

[9] https://en.wikipedia.org/wiki/Integrated_modular_avionics

therefore compete for execution or proceed in a given order) or cannot (and therefore must never compete for execution).

Allocated functions can have a different rate than the process they belong to, in order to express a sampling rate for example. This is given as an input of CHESS model. This rate is an integer *k* meaning that the allocated function has a period equals to *k*P*, where *P* is the period of the process it belongs to. By default, this rate divider is set to one.

Moreover, functions can have an offset, that is counted with respect to a pre-determined instant in time. It is used in order to spread the functions executions over the time and avoid WCET to be particularly important and be responsible of a bad performance. In the example below for example: *funct 0 to 3* are part of *Arinc process*, which has a period equal to the MIF (MInor Frame) – note it is not always the case and can be a multiple of the MIF-. *Funct 0, funct 1* and *funct 2* functions have a rate divider equal to four, meaning the period equals four MIFs. *Funct 3* function has a rate divider equal to eight. Whereas *funct 1* has no offset, funct 0 and *funct 2* have an offset equal to one MIF and *funct 3* has an offset equal to three MIFs. In Figure 8, a colored box means that the corresponding process or the function is executed during the corresponding MIF, note that there can be other processes (not represented in the figure) executing during the same MIF. Offsets are represented with black arrows and are all counted from the thick vertical bar, that could be represented the beginning of the MAF (Major Frame) for example. These offsets will be calculated by a generation tool and will serve to refine schedulability analysis.

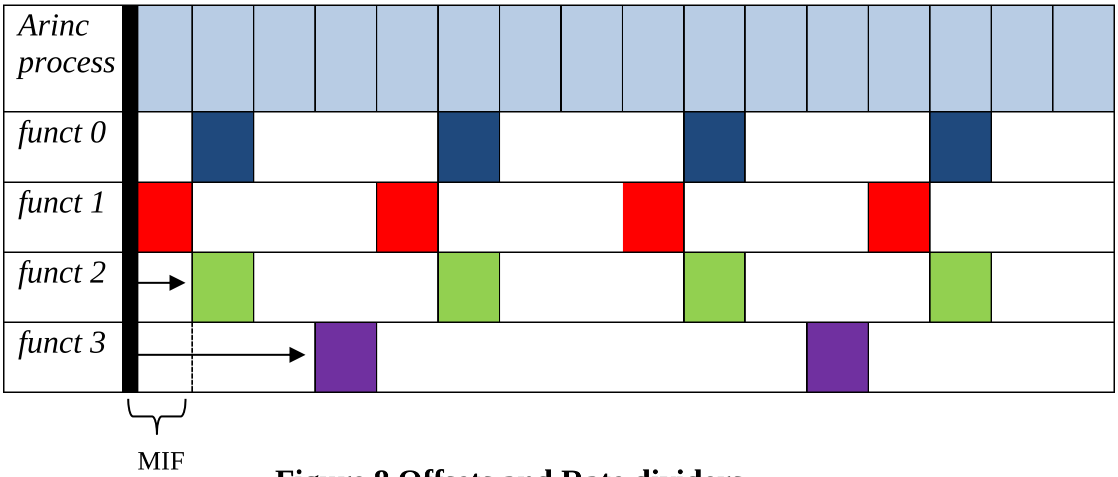


Figure 8 Offsets and Rate dividers

Another feature of interest is added as it enables identifying groups of functions of a same process that need to be run at the same task activation. This information will be used to generate the offset as defined before. Assuming the offsets from the figure above have been generated, one could expect that the CHESS user had grouped together *funct 0* and *funct 2*.

The transformation/generation is needed in order to take into account CHESS model information and generate the MAST++ input model, which has to carry the offset and priority for each of the allocated tasks. Among this information that is inserted by the user: the information concerning the group of operations that can run at the same process activation, the precedence relations between functions, and their rate divider.

In the following sections, details about Avioncs stereotypes of the component model are provided.

13.2 Functional Partitions

An IMA partition is defined in CHESSML as a UML Component stereotyped as FunctionalPartition. The stereotype contains extra-functional properties of the partitions about budget, scheduling ordering and utilization.

13.3 ARINC Processes

New domain specific stereotypes are introduced on top of existing component features to capture ARINC-653 process definition with a single provided operation decorated with a periodic/cyclic attribute, and a number of provided operations. These are private to the component, that is to say not callable by other components. The periodic/cyclic operation may have an offset. The private ones must be able to have priorities and precedence relations which will be expressed between groups of function. This information will be used to compute some offsets, as presented in the detailed description from below:

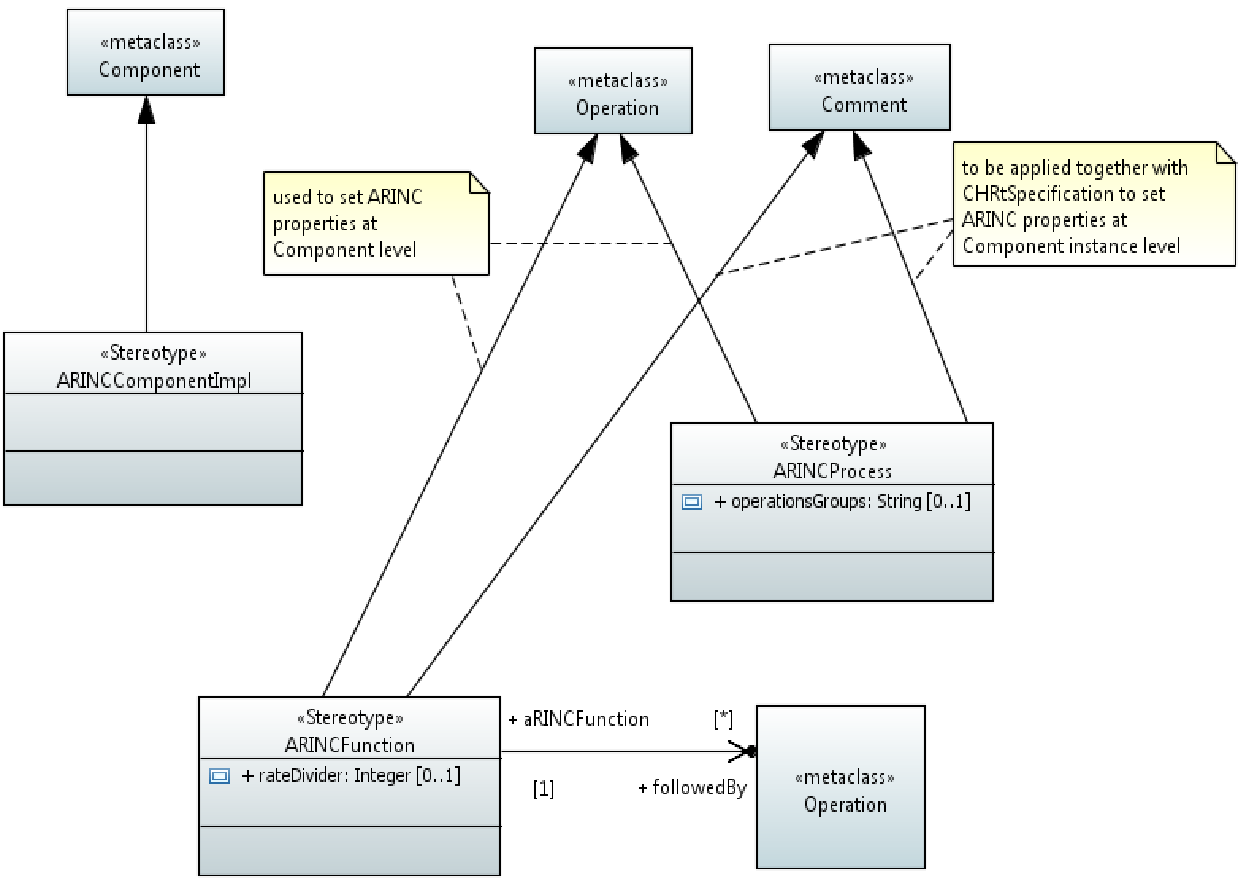


Figure 9: CHESS profile for ARINC processes

- <<ARINCComponentImpl>>: it extends the CHESS <<ComponentImplementation>> in order to add ARINC process semantic. It is available in the extra functional view.

  ARINCComponentImpl comes with the following constraints:

  - The extended Component must be stereotyped as CHESS::ComponentImplementation
  - An <<ARINCComponentImpl >> must define one public operation stereotyped with <<ARINC_Process>> (see below in the text)
  - An <<ARINCComponentImpl >> must define a set (>=1) of <<ARINCFunction>> operations (see below in the text)

- <<ARINCProcess>>: it extends UML::Operation, UML::Comment, and is used to map to the concept of ARINC Process, as collection of operations. It is available in the extra functional view.

  ARINCProcess comes with the following properties:

  - operationsGroups: it defines how the operations which need to be executed when the <<ARINC_Process>> operation is scheduled are grouped; i.e. it defines which operation can run with which other. If an operation is not mentioned in any group than it may be added to any of them during the schedule generation. The string has to be compliant with the given syntax:

    - <expr>:: "(" <operation_list> ")" ["," "(" <operation_list> ")"]
    - <operation_list> ::= <operation> ["," <operation_list>]
    - <operation> ::= <string>

    e.g. (op_1, op_4), (op_3, op_2)

    Precedencies between groups of operations can be derived from the precedencies available between the operations themselves (see <<ARINCFunction>>).

  ARINCProcess comes with the following constraints:

  - The owner of ARINCProcess must be an ARINCComponentImpl. The operations listed in the operationsGroups must be ARINCFunction of the same owning component or external operations (tagged as passive or protected) available for the owning component through required ports.
  - ARINCProcess must not have any implementation specified; the behaviour is the “composition” of the behaviours of the operations that can be scheduled during the ARINCProcess activation

- For ARINCComponentImpl instance, an ARINCProcess operation should have a CHRtSpecification specifying:
    - The occurrence pattern occKind, i.e. information regarding the arrival time, eventually with the phase (i.e. offset) (default is zero); According to MARTE specification, the arrival time can be specified as an expression string making reference to variables, e.g.: "Periodic(period=(value= $MAF + 10), (unit=ms))"
    - The priority

  The WCET is derived as the sum of the operations that can be scheduled during the ARINCProcess activation.

- ARINCFunction extends: UML::Operation, UML::Comment; it represents an operation of the ARINCComponentImpl which maps to an ARINC function. It is part of the ARINCProcess defined in the owning ARINCComponentImpl. It is available in the extra functional view.

  ARINCFunction comes with the following properties:

    - followedBy: Operation[0..*] the operations that must run after
    - rateDivider: Integer[0..1] rate divider of the period associated to the ARINCProcess operation defined in the owning component
    - offset: the phase associated will be calculated during model transformation –from the previous relation- and directly used to set for MAST++ the "Phase" value of the External_Event of Transaction.

  ARINCFunction comes with the following constraints:
    - ARINCFunction as extension of Comment must be used together with the CHESS\MARTE CHRTSpecification stereotype to provide information for an ARINCComponentImpl component instance.
    - The owner of an ARINCFunction as extension of Operation must be an ARINCComponentImpl.
    - An ARINCFunction cannot be invoked by the public periodic operation of the same owner component, or by other ARINCFunction operations.
    - An ARINCFunction cannot appear in a provided interface of the owning component.
    - An ARINCFunction can only call protected or passive operations available through required interfaces of the owning component.
    - The operations appearing in the followedBy and precededBy lists must be ARINCFunction of the same owning component or external operations (tagged as passive or protected) available through required ports.

- For an ARINCComponentImpl instance, an ARINCFunction operation should have a CHRtSpecification specifying:
  - the passive access mechanism, i.e. " CHRtSpecification .protection" = "concurrent"
  - the WCET
  - the phase (i.e. offset) (default is zero)
    - According to MARTE specification, phase can be specified in the context of the periodic activation pattern, e.g.:
      - CHRtSpecification .occKind= "Periodic(phase=(value= 10), (unit=ms))"
    - The value for phase is automatically derived in CHESS
- The ARINCFunction inherits the information about its periodic activation from the ARINCProcess periodic operation of the owning component, on which the rateDivider is applied.

The updated transformation to the analysis tool[10] input model is in charge of generating the offsets that permit a well-balanced workload over the different MIFs. Another role is to generate the priorities for its operations. This is a unique integer that has to take into account both the priority inherited from the owning process and the relationship of precedencies that is given by the “followedBy” information.

13.4 Restriction on the Deployment

In case of FunctionalPartition, the software to hardware allocation has to be compliant with the following rules:

- ARINC processes has to be allocated to Functional Partitions by using the MARTE stereotype <<Assign>> already used in CHESSML for assigning software components to hardware components (as showed in Annex 0)

- FunctionalPartitions have to be allocated to cores (by using the MARTE Assign construct) and eventually to MemoryPartitions (if the latter are available in the deployment view)

14 Requirement

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

14.1 Entities

14.2 Requirement

From SysML::Requirements

[10] MAST++

14.3 Derived Reqt

From SysML::Requirements

14.4 Satisfy

From SysML::Requirements

References

15 References

[CHESS] CHESS Toolset Guide

[SAFEC] L. M. Barbara Gallina, «SafeConcert: a Metamodel for a Concerted Safety Modeling of Socio-Technical Systems,» in *5th International Symposium on Model-Based Safety and Assessment*, 2017.