SESAME
SECURE AND SAFE MULTI-ROBOT SYSTEMS

Project Number 101017258

D4.2 Safety-Targeted ODE and EDDI specification
D5.2 Security-Targeted ODE and EDDI specification

Version 1.0
30 June 2022
Final

Public Distribution

University of Hull, Fraunhofer IESE and FORTH

Project Partners: **Aero41, ATB, AVL, Bonn-Rhein-Sieg University, Cyprus Civil Defence, Domaine Kox, FORTH, Fraunhofer IESE, KIOS, KUKA Assembly & Test, Locomotec, Luxsense, The Open Group, Technology Transfer Systems, University of Hull, University of Luxembourg, University of York**

PROJECT PARTNER CONTACT INFORMATION

Aero41 Frédéric Hemmeler Chemin de Mornex 3 1003 Lausanne Switzerland E-mail: frederic.hemmeler@aero41.ch	**ATB** Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany E-mail: scholze@atb-bremen.de
AVL Martin Weinzerl Hans-List-Platz 1 8020 Graz Austria E-mail: martin.weinzerl@avl.com	**Bonn-Rhein-Sieg University** Nico Hochgeschwender Grantham-Allee 20 53757 Sankt Augustin Germany E-mail: nico.hochgeschwender@h-brs.de
Cyprus Civil Defence Eftychia Stokkou Cyprus Ministry of Interior 1453 Lefkosia Cyprus E-mail: estokkou@cd.moi.gov.cy	**Domaine Kox** Corinne Kox 6 Rue des Prés 5561 Remich Luxembourg E-mail: corinne@domainekox.lu
FORTH Sotiris Ioannidis N Plastira Str 100 70013 Heraklion Greece E-mail: sotiris@ics.forth.gr	**Fraunhofer IESE** Daniel Schneider Fraunhofer-Platz 1 67663 Kaiserslautern Germany E-mail: daniel.schneider@iese.fraunhofer.de
KIOS Maria Michael 1 Panepistimiou Avenue 2109 Aglatzia, Nicosia Cyprus E-mail: mmichael@ucy.ac.cy	**KUKA Assembly & Test** Michael Laackmann Uhthoffstrasse 1 28757 Bremen Germany E-mail: michael.laackmann@kuka.com
Locomotec Sebastian Blumenthal Bergiusstrasse 15 86199 Augsburg Germany E-mail: blumenthal@locomotec.com	**Luxsense** Gilles Rock 85-87 Parc d'Activités 8303 Luxembourg Luxembourg E-mail: gilles.rock@luxsense.lu
The Open Group Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium E-mail: s.hansen@opengroup.org	**Technology Transfer Systems** Paolo Pedrazzoli Via Francesco d'Ovidio, 3 20131 Milano Italy E-mail: pedrazzoli@ttsnetwork.com
University of Hull Yiannis Papadopoulos Cottingham Road Hull HU6 7TQ United Kingdom E-mail: y.i.papadopoulos@hull.ac.uk	**University of Luxembourg** Miguel Olivares Mendez 2 Avenue de l'Universite 4365 Esch-sur-Alzette Luxembourg E-mail: miguel.olivaresmendez@uni.lu
University of York Simos Gerasimou & Nicholas Matragkas Deramore Lane York YO10 5GH United Kingdom E-mail: simos.gerasimou@york.ac.uk nicholas.matragkas@york.ac.uk	

DOCUMENT CONTROL

Version	Status	Date
0.1	Initial draft with outline	25 Apr 2022
0.2	Full initial draft completed	10 May 2022
0.3	Updated with latest metamodel changes	31 May 2022
0.4	Combined with D5.2 security document to obtain merged version	6 June 2022
0.9	Almost final version incorporating feedback from BRSU, KUKA, and Locomotec	29 June 2022
1.0	Final version with last edits from IESE	30 June 2022

TABLE OF CONTENTS

TABLE OF FIGURES

EXECUTIVE SUMMARY

This document sets out the specification of the Open Dependability Exchange (ODE) metamodel and the new modifications and extensions proposed as part of the SESAME project. The ODE was originally created to describe design-time Digital Dependability Identities (DDIs); however, to adapt the concept to runtime and support Executable DDIs (EDDIs), various additions to the metamodel were required.

The changes are described in this report and broadly consist of the following:

- An event monitoring and action response framework to support real-time system feedback;
- The addition of Conditional Safety Certificates (ConSerts) to support the exchange of dependability guarantees and demands within multi-agent systems;
- Updated causal models for safety and security analysis, including extensions to fault trees, attack trees, state machines, and the addition of Bayesian networks (all of which support events and actions to allow runtime evaluation based on real-time data);
- Proposal for a new package to support situation-aware dynamic risk analysis at runtime;
- Preliminary ideas for assessment of ML reliability and robustness at runtime.

This updated ODE will allow the creation or modification of tools to generate EDDIs from design-time models that can then be executed at runtime.

LIST OF ABBREVIATIONS

AADL	Architecture Analysis & Description Language
ADL	Architecture Description Language
AI	Artificial Intelligence
ALARP	As Low As Reasonably Possible
ASIL	Automotive Safety Integrity Level
BE	Basic Event (i.e., root cause of a fault tree)
CAE	Claims, Arguments, Evidence framework
CCA	Common Cause Analysis
CCF	Common Cause Failure
CFT	Component Fault Tree
CMC	Component Markov Chain
CNN	Convolutional Neural Network
DAL	Design Assurance Level
DDI	Digital Dependability Identity
DNN	Deep Neural Network
DRA	Dynamic Risk Assessment
DSPN	Deterministic Stochastic Petri Net
ECDF	Empirical Cumulative Distribution Function
EDDI	Executable Digital Dependability Identity
FMEA	Failure Modes & Effects Analysis
FTA	Fault Tree Analysis
GSN	Goal Structuring Notation
HARA	Hazard Analysis & Risk Assessment (or Hazard And Risk Analysis)

HAZOP	Hazard & Operability Study
HiP-HOPS	Hierarchically performed Hazard Origin & Propagation Studies
IDS	Intrusion Detection System
MAS	Multi-Agent System
MBSA	Model-based Safety Analysis
ML	Machine Learning
MRS	Multi-Robot System
MTTF	Mean Time To Failure (or MTBF: Mean Time Between Failure for repairable failures)
MTTR	Mean time to repair
NN	Neural Network
OAS	Open Adaptive System
ODE	Open Dependability Exchange
OOD	Out of Distribution
PRA	Probabilistic Risk Assessment (or Analysis)
SACM	Structured Assurance Case Metamodel
SAML	Safety Analysis Modelling Language
SDM	Safety Domain Model
SEFT	State/Event Fault Tree
SIL	Safety Integrity Level
SINADRA	Situation-Aware Dynamic Risk Assessment
UAS	Unmanned Aircraft/Airborne System
UW	Uncertainty Wrapper
XAI	Explainable AI

1. INTRODUCTION

1.1 DEVELOPMENT OF THE ODE

The Open Dependability Exchange (ODE) metamodel was first created in the Horizon 2020 DEIS project[1] to define Digital Dependability Identities (DDIs). DDIs are self-contained, analysable, and composable models that contain all the information necessary to uniquely describe the dependability characteristics of a component or system. The motivation was to capture all the required dependability-related artefacts related to a system/component in a standardised, machine-readable, and model-based form. This includes a broad spectrum of artefacts across the system lifecycle: requirements, system design architecture models, hazard and risk analyses, safety models such as fault trees and Markov models, and safety argumentation.

Although DDIs are synthesised at design-time, the reasoning was that DDIs would be usable at runtime to support dependability assurance activities of distributed, heterogeneous cyber-physical systems — where the operating environment and even the overall system architecture cannot be known a priori, such as platooning of self-driving vehicles. Because it is not possible to obtain a full understanding of such systems purely at design-time, DDIs allow aspects of the assurance process to be moved at runtime instead. Cooperating systems and components could exchange DDIs to assess whether demanded safety requirements are met by the services provided. DDIs can then be composed hierarchically or otherwise connected to other DDIs to form a description of an entire system (or even system of systems), theoretically allowing for automated integration of such systems in the field.

By encapsulating all the necessary dependability information into a single standardised entity, DDIs facilitate the exchange and evaluation of dependability characteristics for a system or component. They also serve as living assurance cases.

Assurance cases — also known as just safety cases when that is the only dependability property under consideration — are a form of structured argumentation that aims to provide justification, with supporting evidence, about why the system is believed to be safe (or secure, or otherwise dependable). In other words, an assurance case is meant to provide proof that the dependability requirements have been met.

A DDI does this by combining everything as part of a single entity: the requirements, the assurance case argumentation, the system model itself for context, and any analyses that provide evidence to support the argumentation.

However, typical assurance cases are static entities that are typically produced near the end of the design process as part of validation & verification and then simply stored away. A DDI, on the other hand, is designed to be more than a simple static artefact, and is instead meant to be capable of updating and evolving over the system lifecycle.

An example of the overall structure of a DDI can be seen in Figure 1 below, the various major elements involved. On the left is an argumentation structure: a claim that a safety requirement has been met, linked to a strategy for demonstrating that claim, which in

[1] https://www.deis-project.eu/

turn links to another claim with supporting evidence (in this case, results of a safety analysis).

On the right are the elements that make up that evidence:

- The Domain package specifies the relevant safety standards in play.
- The Design package is used for modelling the design architecture of the system.
- The HARA package contains the Hazard Analysis & Risk Assessment for the system.
- The FailureLogic package contains information about the failure behaviour of the system, while Measure specifies what measures were taken to mitigate the failures/hazards identified.

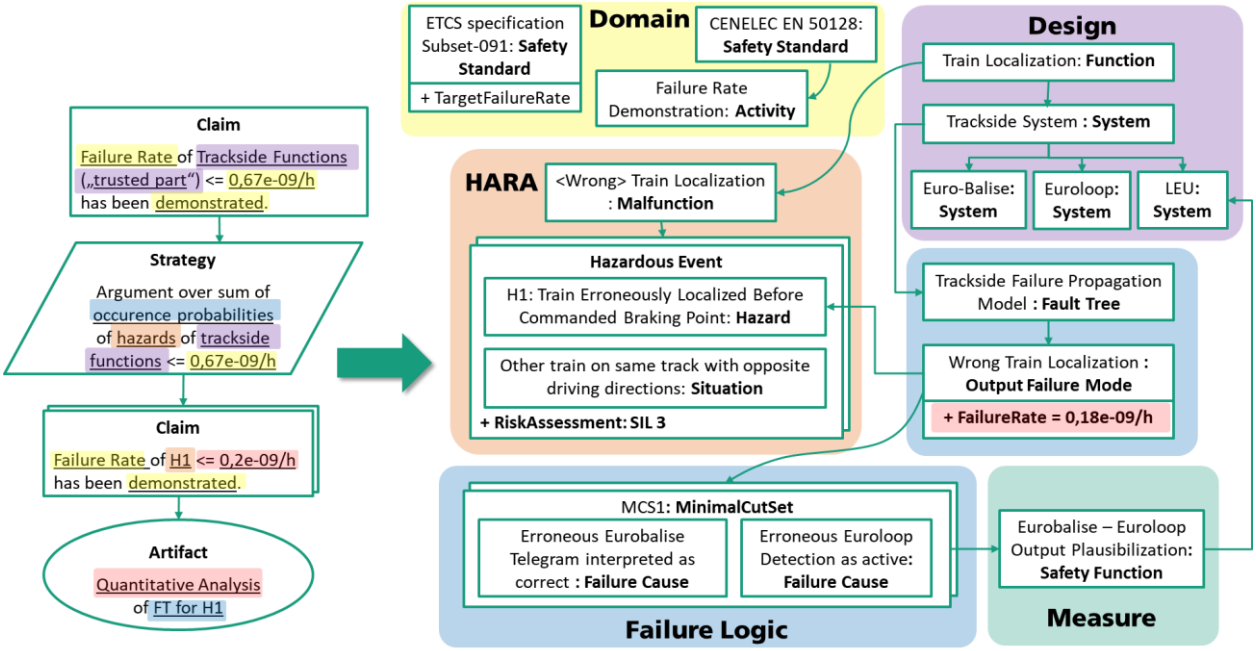


Figure 1 - Illustrative DDI for a dependability assurance case

The ODE was developed to support the various models that may comprise a DDI. It contains two major constituent elements: the assurance case metamodel (which is a version of SACM, the Structured Assurance Case Metamodel[2]), and the product metamodel (which describes the system and its dependability characteristics).

Developed by the Object Management Group, SACM is an attempt to formalise the process of developing safety argumentation and assurance cases. It defines an assurance case as a set of claims, arguments, and supporting evidence to prove justify that the safety requirements for a system have been met.

SACM consists of five main elements.

[2] https://www.omg.org/spec/SACM/2.2/About-SACM/

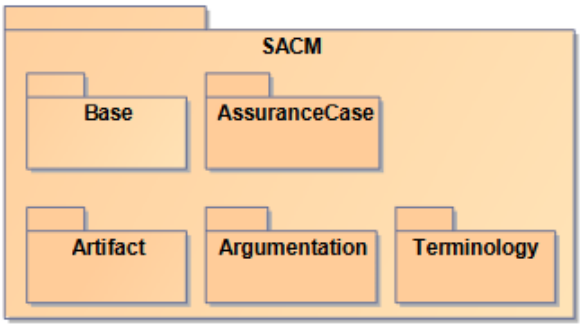


Figure 2 - SACM elements (from [1])

The AssuranceCase element encapsulates all the concepts necessary to produce an assurance case, like a top-level container. This is further supported by the Argumentation component (for making claims and arguments), the Artifact component (representing concepts used in defining evidence for the arguments), and the Terminology component (which helps to define the vocabulary used to express system properties and characteristics). The Base component, meanwhile, encapsulates all basic entities such as multi-language strings.

As the ODE is effectively a superset of SACM and makes no changes to it, SACM will not be discussed in detail in this document. For further information, please consult the OMG specification (https://www.omg.org/spec/SACM/2.2/About-SACM/).

The ODE product metamodel, on the other hand, is considerably larger and adds the various entities needed to describe a system and model its dependability. As originally specified, it contains the following packages:

- The Base package is the base for everything else and includes fundamental information like names, descriptions, and identifiers.
- The Design package is used to model the system architecture. It defines entities to describe systems, functions, components, interface ports, and connections between them all.
- The Dependability package relates to dependability requirements, standards, and risk analyses — in particular, measures enacted to support or ensure dependability. It also has sub-packages that relate to safety standards etc.
- The Requirements package is a sub-package of Dependability and focuses on dependability requirements (both safety and security). A requirement links to the related failures and hazards as well as mechanisms to address them.
- The HARA or Hazard Analysis & Risk Assessment package supports the modelling of hazards, malfunctions, and associated risks.
- The Failure Logic package contains all the elements required to support failure modelling and safety analysis. It contains sub-packages for the various supported analysis methods: Fault Tree Analysis (FTA), Failure modes and Effects Analysis (FMEA), and Markov analysis.

- The TARA or Threat Analysis and Risk Assessment package supports description and modelling of security threats and analyses such as attack trees.

- Finally, there is an additional small package for Integration (to integrate the packages together).

Further information on the existing ODE packages can be found on the official ODE Github repository[3]. Some additional description of referenced packages can also be found in the appendix (Section 0). Finally, additional information about both DDIs and the ODE can be found in deliverable **D4.1: Safety Analysis Concept & Methodology for EDDI Development.**

1.2 MOTIVATION FOR ODE EXTENSIONS IN SESAME

The overall goal of SESAME is to develop an open, modular, configurable, model-based approach for systematic engineering of dependable multi-robot systems (MRS). The unique challenges of MRS with regard to dependability — namely, complexity, intelligence, and autonomy — all pose obstacles to that goal, and any solution needs to address them as a cohesive whole.

In order to achieve this, a common knowledge repository is required: a model (or set of models) that encompass all the necessary dependability activities and information. These models need to be compatible with each other, modular (to support the open and distributed nature of an MRS), and be applicable both at design time and at runtime.

The Executable Digital Dependability Identity (EDDI) is intended to be the answer to this problem. As an extension of the DDI concept, they are model-based artefacts that contain all the required dependability information about a given system or component — such as safety and security hazards, their potential causes, effects, and possible corrective actions, as well as safety argumentation and information about the system architecture itself. They should also support any relevant dependability activities, whether that be safety analyses, allocation of requirements, or synthesis of safety argumentation. Unlike DDIs, however, EDDIs are intended to be fully executable at runtime, capable of communicating and adapting to changing circumstances to help ensure continued safe operation.

As with DDIs, they are meant to be hierarchical and composable. Unlike DDIs, EDDIs are executable and can adapt themselves to changing circumstances at runtime, e.g. by linking or "plugging in" the EDDI provided by a required service provider to the EDDI of the consumer of that service. If the service provider changes, the EDDI can instead link dynamically with the new one.

Figure 3 illustrates this idea by showing how a EDDI for an overall system (on the left) can be composed of EDDIs for its dynamically-changing subsystems (e.g. S1) which are in turn composed of individual components (A, B, C etc.). Although this is just an abstract example, in the general case, any EDDI can contain other EDDIs as required, and such composition can form dynamically at runtime as well as being arranged statically at design time.

[3] https://github.com/Digital-Dependability-Identities/ODE

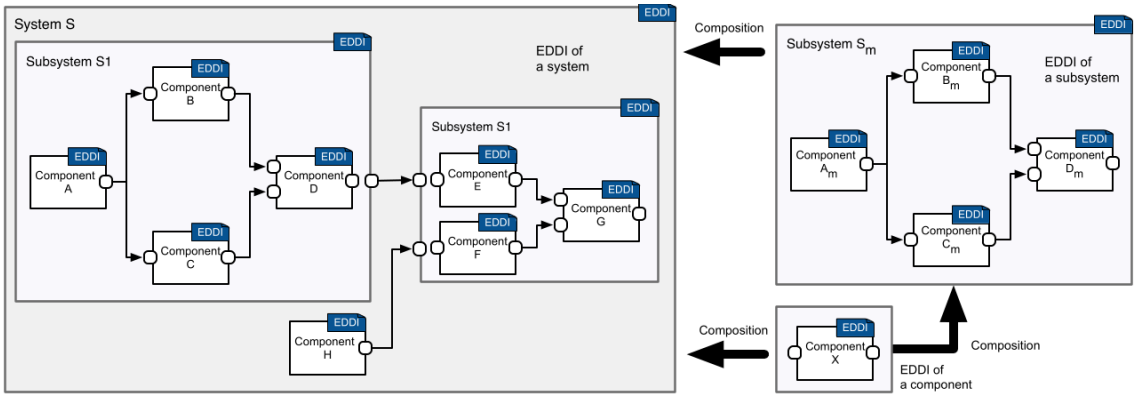


Figure 3 - Composition of DDIs

To support the EDDIs, new extensions to the ODE are necessary that focus particularly on dynamic or runtime capabilities. This includes:

- Generic concepts to support runtime execution and communication with the host system, described in Section 2;
- New and/or extended safety analysis models targeted at dynamic behaviour, including state machines and Bayesian networks (covered in Section 3);
- Additional concepts to support security modelling and threat analysis (which constitutes the merged D5.2 deliverable);
- Modelling support for dynamic risk analysis & evaluation, including further behaviour and environment modelling and preliminary support for safety of machine learning (see Section 5).

Work on extending the ODE was an iterative process, involving a series of workshops between partners from WP4, WP5, and WP7 in which changes were proposed, reviewed, and either modified or accepted. While the ODE specification set out in D4.2 / D5.2 is accepted and intended to be stable, it does not rule out further changes that may be made later in the project as required to support the EDDI concept.

Note that although this work on extending the ODE is officially divided across WP4 (safety) and WP5 (security), and touched on by WP7 (runtime), in practice it is difficult to establish a clear delineation between the three areas. Much of the generic runtime support is applicable to all three work packages, and there are close links between the safety and security models. As such, work on the ODE has been a collective effort by Hull, Fraunhofer IESE, and FORTH independent of work package lines, and it was decided to combine the safety (D4.2) and security (D5.2) ODE specifications into a single merged report.

2. GENERIC RUNTIME SUPPORT

2.1 CONCEPT

One of the major goals of the EDDI is to provide support for execution of dependability-related activities at runtime. This may include executing dynamic risk, safety, or security analyses, re-evaluating guarantees and demands related to dependability requirements, or diagnosis of faults and failure prediction/prevention, amongst others.

In order to achieve this, the EDDI requires some means of communicating with its host system (e.g. a drone or robot). For example, dynamic risk assessment may require information about the operating environment, while feedback from onboard sensors may be needed for fault diagnosis and failure prediction. Equally, there needs to be a mechanism to specify how the EDDI communicates whatever conclusions or results it reaches back to the host.

Furthermore, since EDDIs are expected to be used as part of multi-robot (MRS) or multi-agent systems (MAS), they may need to be able to communicate not just with their hosts but also to other robots present in the wider MRS — or rather, the other EDDIs running on those robots. This allows the rest of the MRS to react to problems detected on one robot and adapt accordingly, e.g. by updating the task allocation to accomplish the same mission with the remaining robots.

While implementation-specific details about how this takes place in the context of the real-world operating environment are out of scope of the ODE metamodel, the ODE does need to be capable of specifying what information is conveyed and when/why. This is accomplished through the use of three main mechanisms: Events, Actions, and Conditional Safety Certificates (ConSerts).

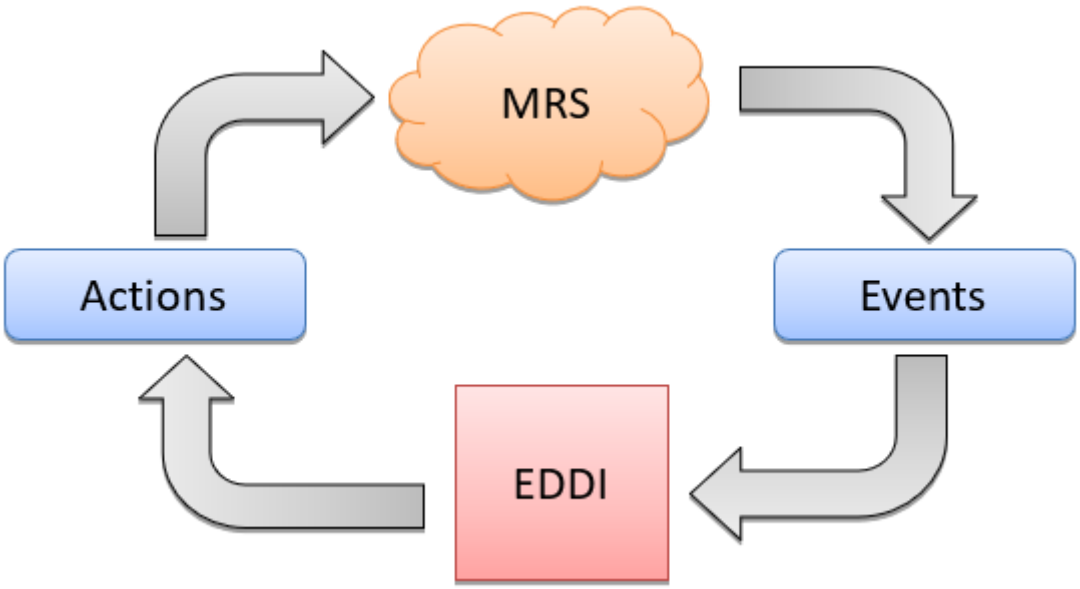


Figure 4 - The Event-Action cycle

Events act as feedback from the host system to the EDDI, allowing it to react to detected failures, anomalous sensor readings, and so on. Actions provide the means to

communicate recommendations and warnings back to the system (or other systems in the MRS). Finally, ConSerts serve as the mechanism to support the exchange of demands and guarantees to ensure that overall safety requirements are met across the entire MRS.

2.2 EVENTS

To model the way in which an EDDI is able to respond to data from the host system, an event-based approach has been adopted. An event is essentially a notification that something has occurred, e.g. a condition being fulfilled, a failure being detected, or a message received etc. In our case, we match these events with actions, which are the actions we wish to make in response to the events.

The resulting modelling entities for Events (and for Actions, covered next) comprise the new ODE::Event package. The event side of the package is intended to describe events and event monitoring processes.

The goal of this package is not to model any and all events that may occur in the operation of a system, but only those which are observable (i.e., can be monitored at runtime and which have a detectable trigger) and which have safety or security implications or which otherwise have important effects relevant to the EDDI. It is also important to note that while a failure might lead to an Event, not all Events are failures.

Events are intended to serve as input to an EDDI at runtime, providing feedback from the system to the executing EDDI safety/security monitor. Events are therefore functionally similar to messages received by the EDDI (and may be implemented in this way), while Event Monitors are the runtime entities that create those messages.

There are different categories of Event that can be distinguished in this context, including (but not necessarily limited to):

- "Condition" or "observed" events, which are conditions that hold over one or more variables/values and which occur when those conditions are observed to be true;
- "Intelligent" or "Machine Learning" events, which are triggered as a result of some kind of decision-making process (such as an ML classification or security intrusion alert) and which may involve some degree of uncertainty;
- "External" events, which are received externally from other EDDIs operating in the wider context (or, potentially, the same EDDI). This may involve information about failures or problems encountered by other robots in the MRS, for instance.

These three categories form subclasses of the primary Event class. However, to be relevant at runtime, we also need to know how to monitor them, which is the purpose of the EventMonitor. Having both allows us to separate the concept (Event) from its implementation (EventMonitor), at least to a degree, in the sense that EventMonitors serve as the interface with the host system and are responsible for converting e.g. sensor data into Event messages that the EDDI can process. This means that the Events can

remain relatively generic and platform-independent, since the implementation-specific details go in the monitor.

The Event metamodel is shown in Figure 5 below. Entities in yellow are part of the Event package; entities in blue are external entities from another package/sub-package; finally, entities in orange are part of the machine learning sub-package and are described in Section 5.3. (Note that IDSEvent is covered further in Section 4)

The Event package depends on the ODE::Base package (for BaseElement) and the ODE::Design package (for System). It is referenced by ODE::FaultTree (Cause), ODE::StateMachine (Transition), ODE::BayesianNetwork (RequiredEvidence), ODE::ConSert (RuntimeEvidence), and by Actions (part of the ODE::Event package), covered next.

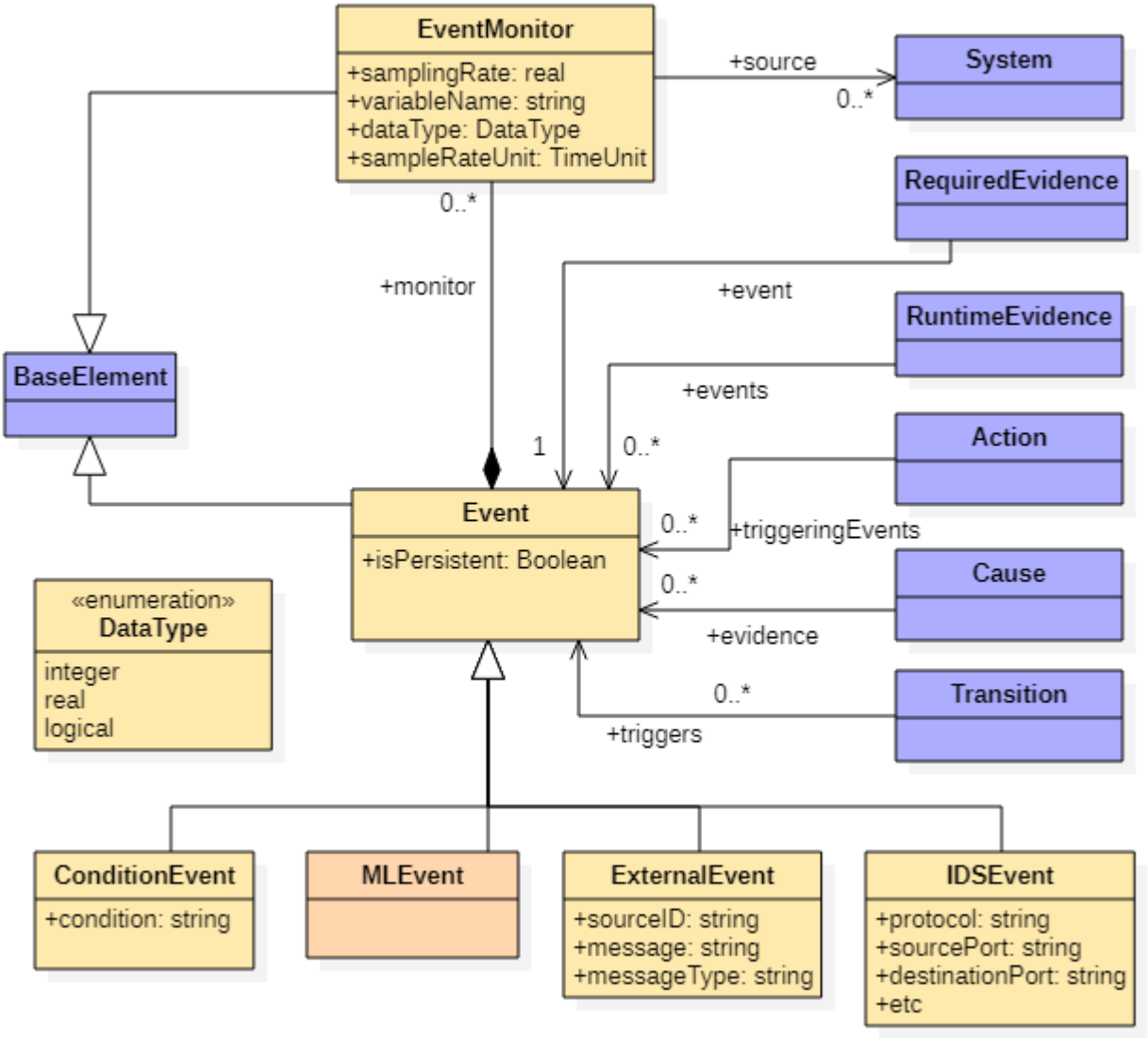


Figure 5 - Event metamodel (colour key described above)

ODE::Event::Event

Represents a generic Event, i.e., a message indicating that something important has occurred. Note that it inherits from **BaseElement**, which provides basic attributes like Name and ID.

- *isPersistent*: A flag that indicates whether the Event is intended to be persistent, i.e., whether once occurred, it stays 'true' — like a latching behaviour — or whether it can switch from false to true and back repeatedly.

- *monitor [0..*]*: The monitor(s) that monitor this Event.

ODE::Event::ConditionEvent

Represents a "condition" event, one that involves conditions that test one or more variables (most likely deriving from onboard sensor readings). When those conditions are observed to be true by the event monitors, the Event occurs. For example, a value being over a certain threshold for a given period of time, or two different readings diverging beyond a given margin of error.

- *condition:* A string containing a formal specification of the event condition (see grammar below).

ODE::Event::MLEvent

Represents a complex event triggered by some machine learning process or component, such as an ML classification or a detected security intrusion. Topmost class of the MLEvent hierarchy.

ODE::Event::ExternalEvent

An message event that originates from elsewhere, such as from another EDDI operating in the wider MRS or from the host platform itself. Such events can be used to e.g. update information about offered safety guarantees or to inform of some system failure. Can be triggered by an **Action**, as described in Section 2.3.

- *sourceID:* An identifier that identifies the source of the event. Note that since the source is external, the ID must be globally unique if it is to be capable of identifying the source.

- *messageType:* A string indicating the user-defined type of the message.

- *message:* A string containing the message being communicated by the other agent.

ODE::Event::IDSEvent

An event raised as a result of a security attack detected by an intrusion detection system. For more information, see Section 4; only a subset of fields are shown here.

- *protocol:* An indication of the protocol in use (e.g. TCP, MAVNet).
- *sourcePort:* The originating port of the intrusion.
- *destinationPort:* The target port of the intrusion.

ODE::Event::EventMonitor

Represents information about how an event will be monitored/detected, including implementation specific details like sampling rates etc.

- *samplingRate:* The sampling rate of the monitor, if relevant.
- *samplingRateUnit*: A TimeUnit indicating the time units of the sampling rate. Note that technically it indicates the reciprocal, e.g. if the TimeUnit is seconds, then the sampling rate is Hertz (1/seconds).
- *variableName:* The corresponding variable name, as used in the condition (for **ConditionEvents**). For example, if the event is triggered when motor temperature is over a given threshold, e.g. "temp > 50", then "temp" is the variable being monitored by this EventMonitor (which is likely to be connected to a thermal sensor of some sort).
- *dataType:* The type of data being sampled, e.g. integer, real, logical (as listed in the DataType enumeration). Note that we only consider single values here; for more complicated cases involving e.g. array data, it would be up to the implementor to decide how to proceed in practice.
- *source[0..*]:* The system/component(s) that provides the source of the information being monitored, e.g. a sensor.

For the ConditionEvents, the condition needs to be specified in a format that can be both parsed at design time (for error checking) and executed at runtime (or readily translated to an executable form, e.g. code or postfix form for easy stack execution). For this purpose, a grammar for the conditions is required. It should encompass all the different types and combinations of conditions that may be encountered:

- Simple Boolean conditions, e.g. "variable X is true";
- Numeric comparisons, e.g. "X > 100";

- Time durations to help filter out spurious readings, e.g. conditions that must hold for a given period of time;
- Logical combinations of conditions (AND, OR, NOT, XOR);
- Basic arithmetic operations (e.g. addition, multiplication, powers);
- Arithmetic functions, e.g. MAX, MIN, ABS, SUM, AVERAGE;
- Potentially also calculus (rate of change, differentiation, integrals etc).

An additional complexity is that at runtime we also need to account for the fact that our knowledge is limited. Therefore, in addition to simple true/false conditions, we also need to accept the possibility of unknown results, meaning we will be employing a three-value logic — TRUE, FALSE, UNKNOWN — instead. UNKNOWN may apply, for example, if a sensor fails to provide a reading, or a function does not yet have enough historical data to provide a result.

The grammar is defined in EBNF[4] as follows:

```
condition =          timed_exp, {"AND" | "OR" | "XOR",
                     timed_exp };

timed_exp =          bool_exp
                     | "TIME", "(", bool_exp, ",",
                     constant, time_unit, ")";

time_unit =          "ms" | "s" | "m" | "h" | "d";

bool_exp =           bool_comp, {"AND" | "OR" | "XOR",
                     bool_comp}
                     | "(", bool_exp, ")";

bool_comp =          not_exp, [("==" | "!="), not_exp];

not_exp =            ["NOT"], bool_val;

bool_val =           comparison
                     | variable
                     | "EVENT", "(", variable, ")"
                     | bool_constant;

bool_constant =      "TRUE" | "FALSE" | "UNKNOWN";

comparison =         expression, rel_op, expression;
```

[4] https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form

```
rel_op =            "==" | "!=" | "<" | ">" | "<=" | ">=";

expression =        term, {["+" | "-"], term};

term =              factor, (["*" | "/"], factor};

factor =            (num_val, ["^", num_val])
                    | unary_function
                    | timed_function;

unary_function =    "ABS" | "SQRT", "(", expression, ")";

timed_function =    func_name,
                    "(", variable, ",", constant,
                    time_unit ")";

func_name =         "MAX" | "MIN" | "SUM" | "AVERAGE"
                    | "DIF" | "INTEG";

num_val =           variable | constant |
                    "(", expression, ")";

variable =          any valid identifier

constant =          any integer or real number
```

This grammar allows for a variety of expressions, including:

- Boolean expressions, e.g. `X == TRUE AND Y == FALSE`
- Conditions that must hold over time, e.g. `TIME(temp > 100, 5s)`
- Negated conditions, e.g. `X == NOT UNKNOWN`
- Comparisons, e.g. `battery_level <= 100`
- Arithmetic operations, e.g. `battery_level / 1000 <= 50`
- Powers, e.g. `x ^ 2 < 100`
- ABS and SQRT, e.g. `ABS(var) > 5`
- Functions that operate over a range of historical values, e.g. MAX, MIN, SUM, AVERAGE, DIF, INTEG etc. The operand indicates the period of time over which values should be used. The precise number of values then depends on the time and the relevant sampling rate.
- Combining all of the above, e.g. `AVERAGE(temp, 5s) > 100 AND TIME(warning == TRUE, 5s)`

Variables are taken to be any combination of numbers and letters, as long as it starts with a letter (and excluding the keywords defined in the grammar), while numeric constants can include numbers in base 10 scientific format (e.g. 1e-5) and specified with either '.' or ',' as the decimal separator.

To connect variables to their sources, EventMonitors are needed. An EventMonitor indicates both the variable name, the source component of the value in question (e.g. a sensor), and a sampling rate (which can be used in conjunction with the time constants to determine the number of historical values that must be stored in a buffer). It may also specify the data type provided by the source, e.g. real values, integers, or logical values (TRUE/FALSE/UNKNOWN).

It is assumed that the source of an EventMonitor must be a System (as defined in the ODE::Design package) rather than a Function, since Functions are more abstract and are realised by Systems.

Events can additionally reference each other via the EVENT construct (the variable operand is assumed to be the name of the event in question). Event status is considered to be TRUE, FALSE, or UNKNOWN and thus can only be used as part of a logical expression, not a numeric one. This allows events to be conditional on the status of other events — including different *types* of event, i.e. allowing ConditionalEvents to depend on the status of External or ML events.

To understand how the three-value logic should behave, please consult the following truth table (where T = True, F = False, and U = Unknown):

Table 1 - Truth table for three-value logic

X, Y	X AND Y	X OR Y	X XOR Y	NOT X
F, F	F	F	F	T
F, T	F	T	T	T
F, U	F	U	U	T
T, F	F	T	T	F
T, T	T	T	F	F
T, U	U	T	U	F
U, F	F	U	U	U
U, T	U	T	U	U
U, U	U	U	U	U

2.3 ACTIONS

While Events serve as input to the EDDI from the system or other agents, there must also be a way for EDDIs to provide feedback to the system or other agents. Actions serve as the primary mechanism for specifying the content of this output.

Actions are intended to be an abstracted form of operations/responses that the EDDI recommends on the basis of diagnosis or decision making, as informed by the internal and external conditions observed across the MRS. Conditions are identified using the Events (and their subtypes) and corresponding Actions can be specified based on MRS capabilities and situational awareness.

The proposed Action metamodel is illustrated below.

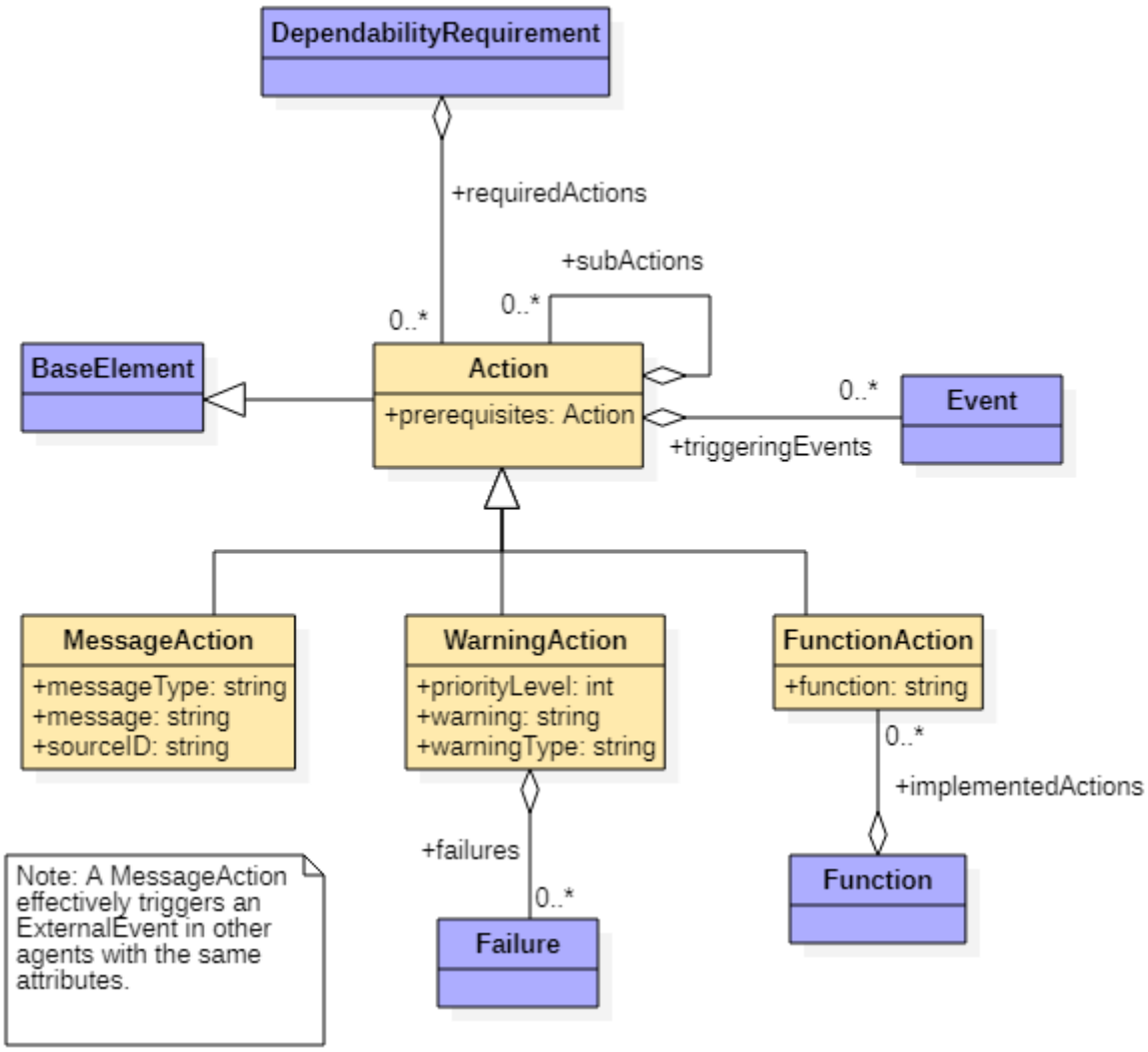


Figure 6 - Action metamodel

ODE::Event::Action

Represents a generic operation that can be performed at runtime.

- *subActions* [0..*]: Allows compound Actions to be created. Ordering of sub-actions can be defined using the prerequisites.
- *prerequisites* [0..*]: Defines a sequence of (sub-)Actions by specifying potential prerequisite Actions that must take place first.
- *triggeringEvents* [0..*]*:* The set of events associated with triggering this action. Note that this is not an explicit causal relationship; the precise triggering logic is embodied by the relevant causal models, i.e., fault trees, Bayesian networks, or state machines. Thus this attribute is indicative only.

Actions are also referenced by **DependabilityRequirements**:

- **ODE::Requirement::DependabilityRequirement**::*requiredActions[0.. *]*: The set of actions required to occur by a given requirement. The interpretation here is that the Actions contribute to the fulfilment of the Requirement (e.g. by establishing a kind of safety mechanism or mitigating action).

Three subclasses of Action exist, each tailored to a different type of response:

ODE::Event::MessageAction

A MessageAction is a generic, informative message. It does not directly trigger any functionality or necessarily indicate a failure. The assumption is that a MessageAction triggers an **ExternalEvent** in other agents (or potentially in the parent EDDI, if appropriately configured), and hence has the same fields.

- *sourceID:* An identifier that identifies the source of the message.
- *messageType:* A string indicating the user-defined type of the message.
- *message:* A string containing the message being sent to other agents.

ODE::Event::WarningAction

Intended to indicate the detection/diagnosis of one or more failures, e.g. in order to display a warning alert to the user.

- *failures* [0..*]: The failures that we are attempting to warn about, if relevant. [0..*] is used in case we wish to emit a generic warning not linked to any particular failure (e.g. that dynamic risk estimation has passed a critical threshold).
- *priorityLevel:* A custom field to express warning priority levels (e.g. low, medium, high priority).
- *warningType:* The type of warning. Also user-defined. Examples could include "Failure Detected" or "Increased Risk" etc.
- *warning:* The content of the actual warning.

ODE::Event::FunctionAction

A recommendation for the system to perform a particular function.

- *function:* The function to perform.

FunctionActions may be referred to by Functions to establish a more concrete link to the design architecture:

- **ODE::Design::Function**::*implementedActions*[0.. *]: The set of actions made available by the implementation of this functionality.

While it may be possible to trigger Actions directly from Events, the primary intention is that Actions arise as a result of logical processes captured by the causal models. Thus an Action may be triggered when a particular fault tree node is determined to be true, or when a state is entered in a state machine, or as part of an OutputEvent in a Bayesian network and so forth (see Section 3 for more).

2.4 CONSERTS

One challenge in ensuring the safety of cooperative automated systems is to deal with uncertainties and unknowns with respect to the cooperating partners. In other words, one might not know what kind of guarantees come along with a certain information or service of a 3rd party system. Unfortunately, the lack of knowledge regarding external services and their safety properties typically leads to worst case assumptions, which in turn severely constrain performance or may even lead to the decision not to use external services or information at all. A straightforward solution to this problem is to enable constituent systems of a MAS to explicitly negotiate their safety-related properties at runtime. This implies that we establish runtime dependability models describing these properties for a (constituent) system and standardize a protocol for their negotiation.

Conditional Safety Certificates (ConSerts) [2] [3] is a concept intended to achieve exactly that. ConSerts operate on the level of safety requirements and are specified at development time based on a sound and comprehensive safety argumentation (e.g. an assurance case). They conditionally certify that the associated system will provide specific safety guarantees. Conditions are related to the fulfillment of specific demands regarding the environment and the fulfillment of the conditions is checked during runtime.

There are some significant differences between ConSerts and static certificates that are owed to the nature of open cooperative systems: a ConSert is not static but conditional; it therefore comprises a number of variants that are conditional with respect to the (dynamic) fulfillment of demands; and it must be available in an executable (and composable) form at runtime.

Conditions within a ConSert manifest in relations between potentially guaranteed safety requirements ("guarantees") and the corresponding demanded safety requirements ("demands"). Demands always represent safety requirements relating to the

environment of a component, which cannot be verified at development time because the required information is not available yet. These demands might directly relate to required functionalities from other components.

On the other hand, evidence can be required beyond that, since safety is not a purely modular property and it cannot be assumed that a composition of safe components is automatically safe. To this end, ConSerts support the concept of so-called Runtime Evidence (RtE) as an additional operand of the conditions. RtEs are a very flexible concept. In principle, any runtime analysis providing a Boolean result can be used. RtEs might relate to properties of the composition or to any context information, e.g. a physical phenomenon such as the temperature of the environment that is safety relevant. Other RtE require dynamic negotiation between components.

In any case, ConSerts must be available at runtime in a machine-readable representation and the systems need to possess mechanisms for composing and analyzing runtime models. Based thereon, a valid safety certificate for the over-all system of systems can be established. ConSerts are a relatively lightweight runtime safety approach and they are not far from traditional safety engineering. The main difference being that unknown context is structured into a series of foreseen variants, which are then specified in a runtime model to be resolved at runtime.

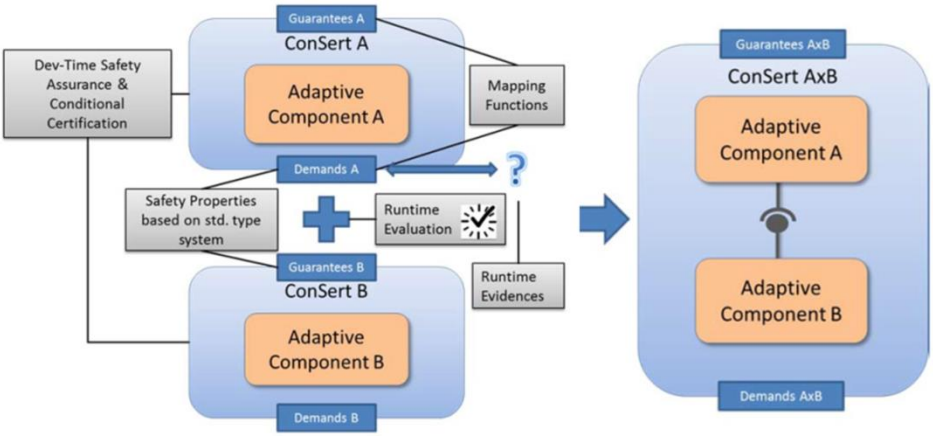


Figure 7 - ConSert Composition Conceptual Overview

The ConSert metamodel describes the elements required to formally describe ConSerts. Overall, three packages are added: ODE::ConSerts, ODE::Service and ODE::Dimension. The Service package extends the ODE:Design package to provide coverage of service-based designs. The ConSert package describes the elements for designing a ConSert. Lastly, the Dimension package enables the specification of runtime attributes which allows the generation of executable ConSerts. An overview with all three of the aforementioned packages and their relations are shown in Figure 8 - Overview of ConSert, Dimensions and Service PackageFigure 8.

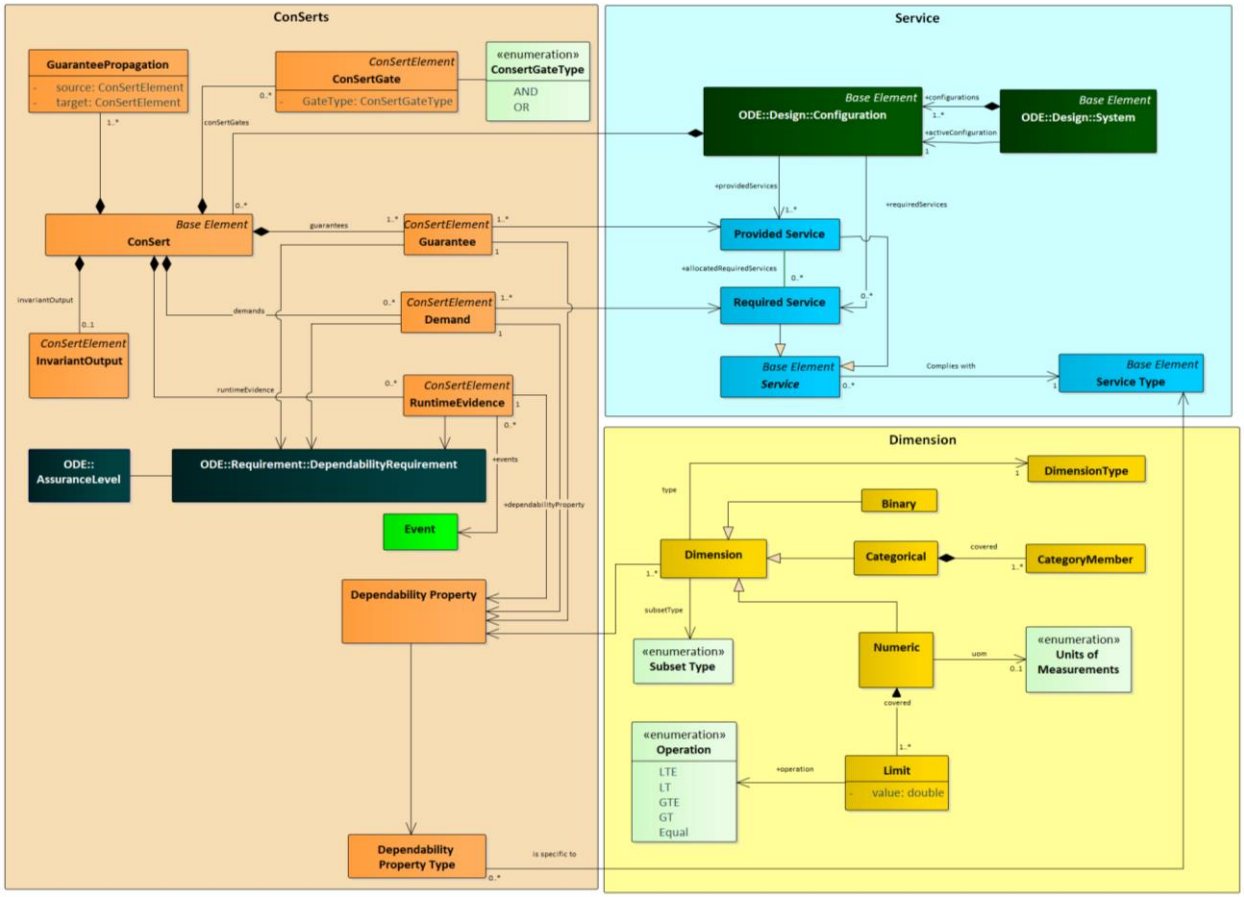


Figure 8 - Overview of ConSert, Dimensions and Service Packages

2.4.1 ODE::ConSert

The ODE::ConSert package extends the ODE metamodel to create a standardized ConSert representation. This package has relations to the ODE::Design, ODE::Requirement, ODE::Service and ODE::Dimension packages. The ODE::ConSert metamodel and its relations to the other packages are shown in Figure 9.

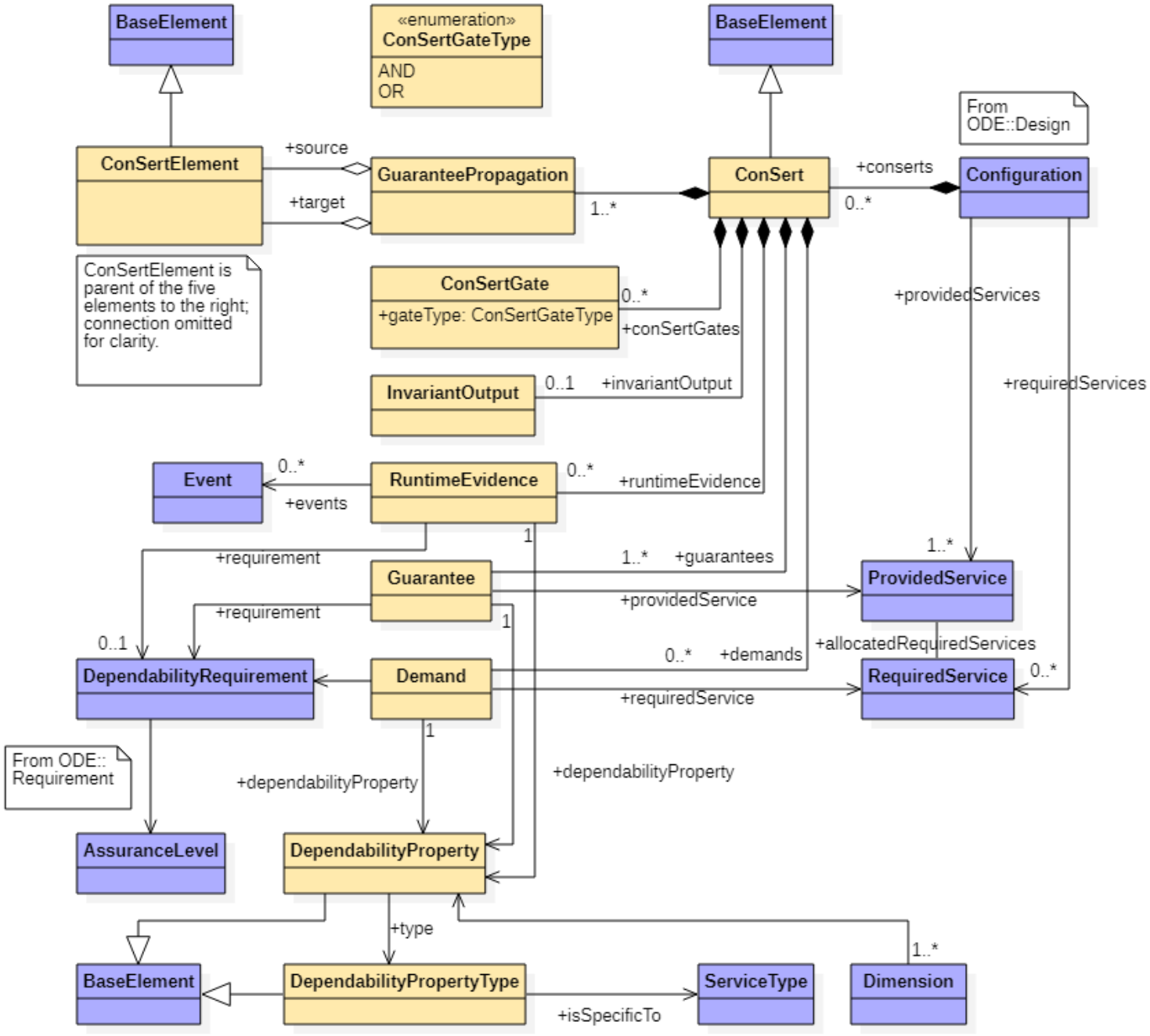


Figure 9 - ODE::ConSert metamodel

ODE::ConSert::ConSert

A Conditional Safety Certificate (ConSert) is a compositional concept used to depict the dependency between **Provided** and **Required Services** of a specific **ODE::Design::Configuration** from a quality point of view. This dependency is established by means of ConSert trees, which are composed of **Guarantees**, **Demands**, **RuntimeEvidence**, **Invariants** and the binding Boolean logic represented by **ConSertGates** and **GuaranteePropagations**.

- *guaranteePropagation[*]:* The GuaranteePropagations associated with the ConSert.
- *invariantOutput[0…1]*: The InvariantOutput, if applicable. More details about invariants can be in the corresponding subsection below (see ODE::ConSert::InvariantOutput).
- *demands[*]:* The demands of the ConSert, i.e., demanded dependability (safety or security) guarantees about external dependencies.
- *guarantees[*]*: The guarantees offered by the ConSert, i.e. dependability guarantees offered to consumers of functionalities covered by this ConSert.
- *runtimeEvidence[*]*: The runtime evidence required by this ConSert to determine whether guarantees or demands can be met.
- *consertGates[*]*: The logic gates that comprise the logical structure of the ConSert.
- *serviceConfiguration*: The associated service configuration.

Notes:

- Given that an **ODE::Design::Configuration** depicts one behavioural variant of the system, only one ConSert should be associated per configuration.

ODE::ConSert::Guarantee

Represents the promise claimed by a system with respect to a **ODE::Requirements::DependabilityRequirement** of a given **ProvidedService.** In order to make use of them during runtime, guarantees need to comply with the **DependabilityProperty** that is associated with the **DependabilityRequirement.** This promise can be seen as the instantiation (i.e. the system's realization) of a **DependabilityProperty**. Therefore, they should concretize any flexibility aspect defined in the **DependabilityProperty**.

- *dependabilityProperty:* The dependability property associated with the requirement.

- *dependabilityRequirement:* The requirement the guarantee must comply with.

- *providedService:* The service covered by this guarantee.

ODE::ConSert::Demand

Represents, for a given system, the minimal requirements in the realization of a **DependabilityProperty** of a given **ODE::Service::RequiredService** that needs to be satisfied during runtime by another system, in order to establish a contract and so enable internal functionalities and dependent **ODE::Service::ProvidedService**.

- *dependabilityProperty:* The dependability property associated with the requirement.

- *dependabilityRequirement:* The requirement the demand must comply with.

- *providedService:* The service covered by this demand.

ODE::ConSert::RuntimeEvidence

Represents a system condition necessary to validate a **Guarantee** during runtime. They are linked to the corresponding **ODE::Requirement::DependabiltyRequirement** that requires the monitoring of this specific **RuntimeEvidence**. Generally, these conditions can only be evaluated at runtime, as they depend on live data. However, they are not limited to it. A **RuntimeEvidence** is either fulfilled or not fulfilled.

- *requirement:* The requirement corresponding to the evidence.

- *events[0..*]:* The **ODE::Event::Event**(s) that provide the evidence, if relevant.

ODE::ConSert::ConSertGate

For the construction of the ConSerts tree, a simplified Boolean logic binding mechanism is a plausible option. This allows defining the logical dependencies between

ConSert constructs, especially between **Guarantees**, **Demands**, **RuntimeEvidence** and other **ConSertGates**.

A **ConSertGate** can be either an **AND** or an **OR** gate. This is specified by the **GateType** field of the **ConSertGate** which is of type **ConSertGateType**. **ConSertGateType** is an enumeration with two types: **AND** and **OR.** The **AND** gate should be interpreted as if the linked runtime evidence or demands contribute to the validation of the guarantees. The **OR** gate is interpreted as an alternative contribution. In this sense, ConSert trees can be depicted as trees having on the top near their roots guarantees, and runtime evidences and demands on their leaves.

- gateType: the type (AND, OR) of the gate

Notes:

- The same semantic applies to the construction of the invariant.

ODE::ConSert::InvariantOutput

An invariant is defined through a Boolean tree that connect the **InvariantOutput** (at its root) to runtime evidence and demands on its leaves. The **InvariantOutput** cannot be connected to **ConSertGuarantees** with **GuaranteePropagation**. This connection is implicitly given for all guarantees.

Invariants represent fundamental conditions necessary for **Guarantees** to be validated during runtime. I.e. the ConSert trees associated with the guarantees will only be validated if the invariant evaluates to true. The semantic of invariants with respect to a ConSert tree is explained in Figure 10

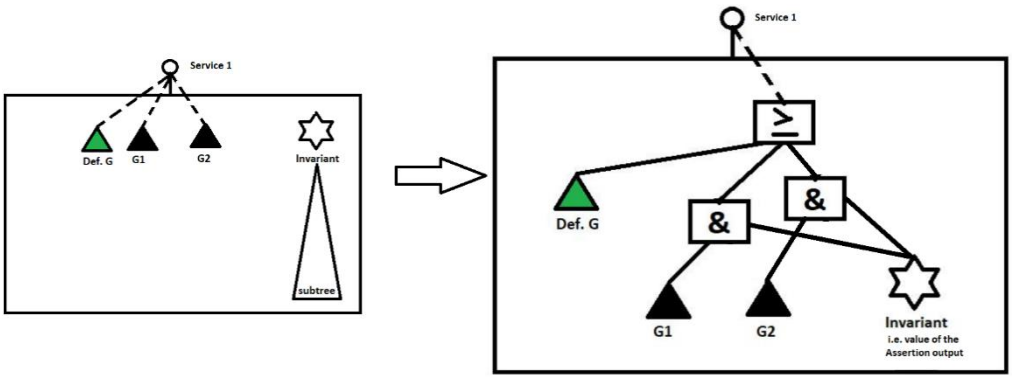


Figure 10 - Invariant semantics

Invariants should be defined in such a way that the condition they represent applies to all **Guarantees** in the **ConSert**, independent of the associated **DependabilityProperty** and **ODE::Service::Service**. If further conditions apply to one or the other guarantee,

these should be defined explicitly as part of the ConSert tree. In this sense, invariants represent only the minimal conditions required to start considering the validation of guarantees at all.

ODE::ConSert::GuaranteePropagation

Connection between ConSert elements i.e. **Guarantees, Demands, ConSertGates, RunTimeEvidence, InvariantOutput** to construct the ConSert tree. The relations have exactly one source and one target ConSert element**.** These relations can connect the ConSerts elements as described below:

Table 2 - ConSert relations

Source	Target
Demand/RuntimeEvidence	**ConSertGate**
Demand/RuntimeEvidence	**Guarantee/InvariantOutput**
ConSertGate	**ConSertGate**
ConSertGate	**Guarantee/InvariantOutput**

ODE::ConSert::DependabilityProperty

The **DependabilityProperty** is associated with a **Demand**, **RuntimeEvidence** and **Guarantee**. Each Demand and Guarantee provides or requires a **DependabilityProperty** that must be fulfilled to composite two **ConSerts**. A **DependabilityProperty** is also associated with an **ODE::Domain::AssuranceLevel** implicitly through the **ODE::Requirement::DependabilityRequirement** linked to a **Demand**, **Guarantee** or **RuntimeEvidence** specifying the required assurance level of the **DependabilityProperty** depending on the domain.

- *dimension[1..*]:* The dimension(s) covered by this property.
- *dependabilityPropertyType:* The type of the property (see below).
- *requirements[1…*]:* Dependability requirements associated with this property.

Note:

- Each **DependabilityProperty** has one assurance level which is formalized with **Categorical Dimensions** (see Section 2.4.3).

ODE::ConSert::DependabilityPropertyType

The **DependabilityPropertyType** reflects a dependability characteristic of the **ServiceType** and therefore shapes a **DependabilityProperty**.

Notes:

- As such types are supposed to be obtained from safety analysis, it is highly probable that non-standard (reusable) types are defined for a **ServiceType**.
- Types are very specific to a given **ServiceType.** This means that a type system could combine the generic types as well as all non-generic types of all service types or, on the contrary, exclude the non-generic types for reasons of simplicity. The completeness of such a type system should be established for each quality domain applicable to the **ServiceType**
- Examples: STRIDE, HAZOP guidewords (too late, too high, …)

2.4.2 ODE::Service

The **ODE::Service** package extends the **ODE::Design** package to support service oriented paradigms. **ConSerts** are associated with system **Configurations** and they have **Guarantees** and **Demands** that are coupled with **Services**.

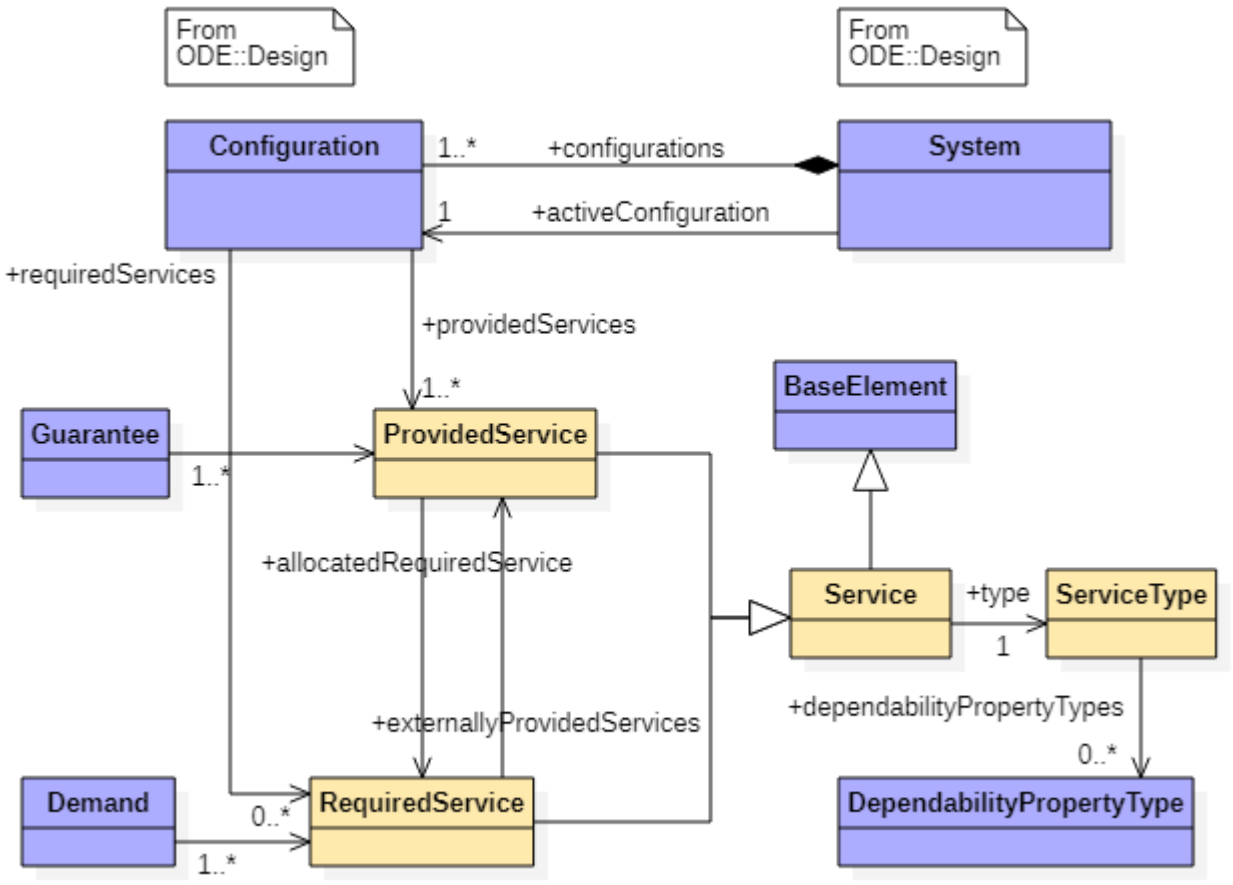


Figure 11 - ODE::Service metamodel

ODE::Service::Service

A **Service** can either be a **RequiredService** or a **ProvidedService**. Each of these **Services** must comply to a **ServiceType** specification. **RequiredServices** and **ProvidedServices** are matched over the **ServiceType.**

- *serviceType[1]:* The associated service type.

ODE::Service::Provided Service

Each **ODE::Design::Configuration** is capable of providing a specified set of **ProvidedServices.** In order to provide some services, a **Configuration** requires services to be provided by other systems. Those **RequiredServices** are allocated to the **ProvidedSystem.** Therefore, an association between **ProvidedServices** to **RequiredServices** is introduced.

- *allocatedRequiredService[*]*: The required services.

ODE::Service::RequiredService

Services that a specific **Configuration** requires to provide their **ProvidedServices.**

- *externallyProvidedServices[*]:* The services provided externally.

ODE::Service::ServiceType

Specifies a portion of the functionality offered by one system and made us of by another system or other users, which only comes to life in the presence of collaboration. It is an abstract construct that defines the core aspects any kind of service type integrates. It includes the **ConSert::DependabilityPropertyType** specification.

2.4.3 ODE::Dimensions & Dependability Property Formalization

In the context of **ConSerts**, Dimensions formalize **ConSert:DependabilityProperty** to instantiate an executable **ConSert**. This ODE::Dimension package is decoupled from ConSerts, such that other package can associate Dimensions to formalize similar elements (e.g. the explicit node state definition and discretization of the Bayesian networks as described in the **ODE::BayesianNetwork** package).

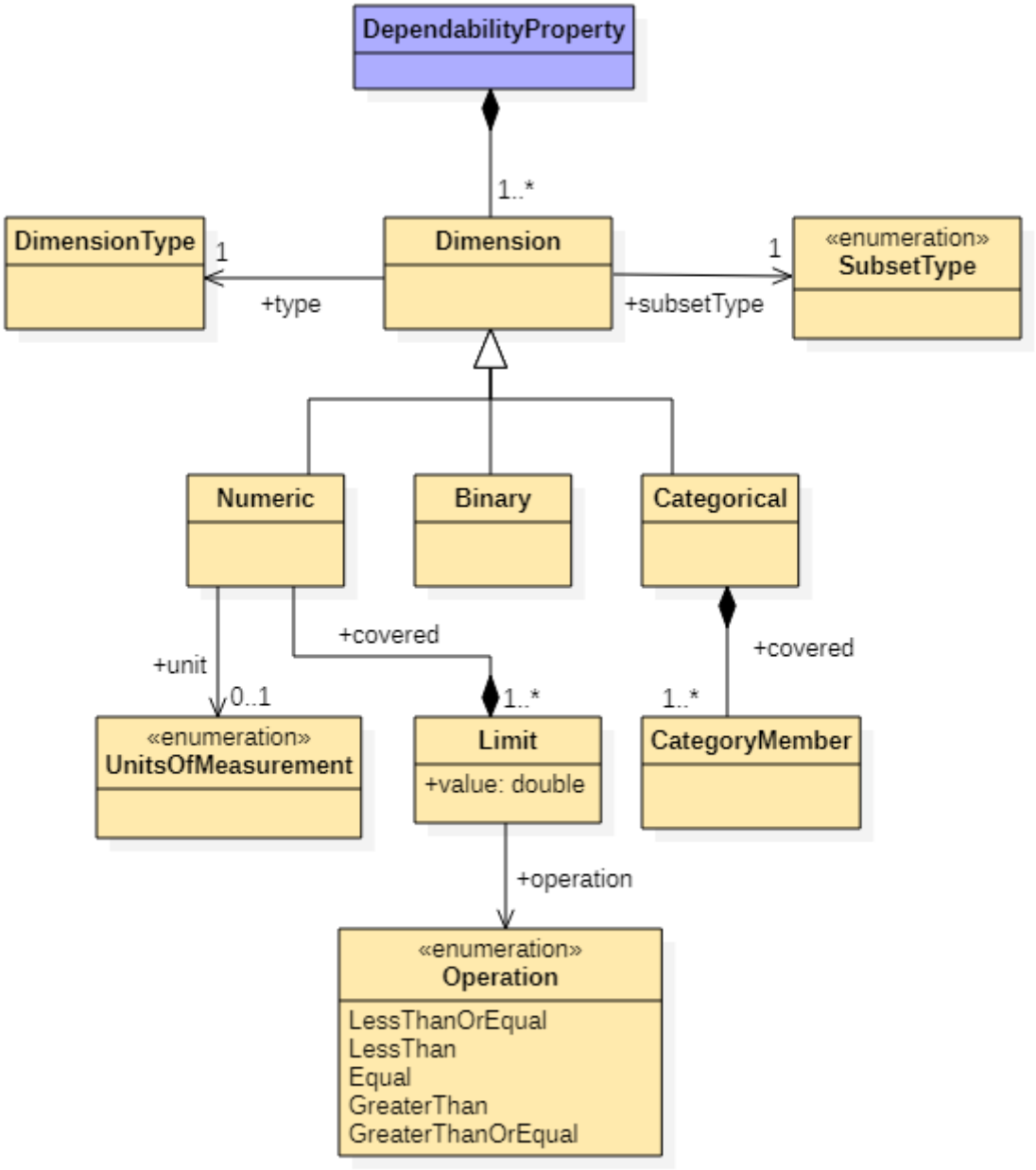


Figure 12 - ODE::Dimension metamodel

ODE::Dimension::Dimension

The matching between **ConSert::Demands** and **ConSert::Guarantees** requires to the specification of properties. These are formally specified as **Dimensions.** Each **Dimension** belongs to a **DimensionType** and specifies a **SubsetType.** A Dimension can be either **Categorical, Binary,** or **Numeric.**

- *subsetType*: The subset type that specifies how to match the dimension (see below).
- *dimensionType*: The type of the dimension (numeric, logical, or categorical).

ODE::Dimension::DimensionType

The **DimensionType** specifies an identifier to ensure, that always the same types of **Dimensions** are matched. For example, a **DimensionType** could specify the velocity of a vehicle, the safe distance to front vehicles, ASIL etc.

ODE::Dimension::SubsetType

The **SubsetType** is an enumeration to specify how two **Categorical** or **Numerical Dimensions** match. The **SubsetType** enumeration is kept abstract to allow **Dimensions** to be used outside of the ConSert context. For ConSerts appropriate **SubsetTypes** could be "Demand in Guarantee" and "Guarantee in Demand" as shown in Figure 13.

Example:

- Demand is subset of Guarantee
    - Demand: Distance to front vehicle > 50 meters
        - type = safe_distance, start = 0, end = 50, inclusive = false, unit = m
    - Guarantee: Distance to front vehicle > 60 meters
        - type = safe_distance, start = 0, end = 60, inclusive = false, unit = m
    - The Guarantee fulfils the Demand if it provides a larger safe distance than the Demand requires. Given the ranges here, the Demand must be a subset of the Guarantee.
- Guarantee is subset of Demand
    - Demand: Maximum speed < 100 km/h
        - type = speed, start = 0, end = 100, inclusive = true, unit = km/h
    - Guarantee: Maximum speed < 90 km/h
        - type = speed, start = 0, end = 90, inclusive = true, unit = km/h
    - The Guarantee fulfils the Demand if it provides a smaller speed than the Demand requires. Given the ranges here, the Guarantee must be a subset of the Demand.

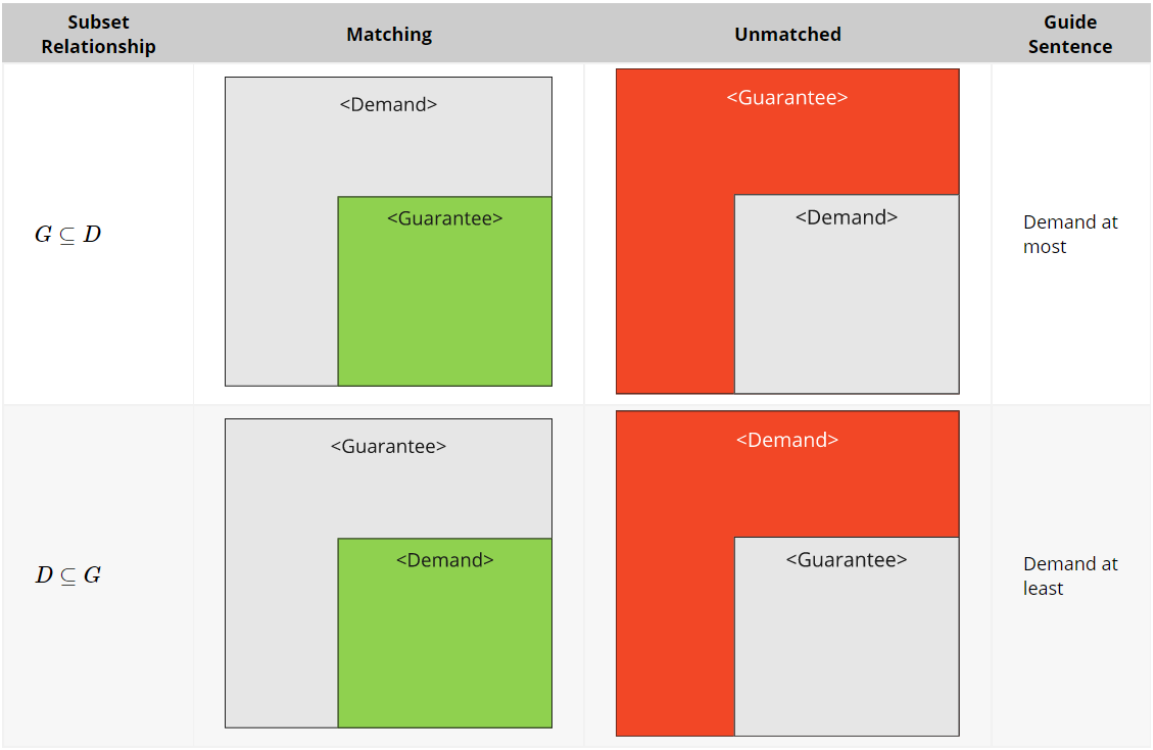


Figure 13 - Dimensions visualization from [OPUS: The Book of ConSerts]

ODE::Dimension::Numeric

A **Numeric** dimension is a **Dimension** with a set of **Limits** specifying a range of values with either an upper or a lower bound. Each **Numeric** dimension specifies a **UnitOfMeasurement**. This enables conversion of different measurement units and allows comparisons between them.

- *covered [1..*]*: The limits covered by this dimension.
- *uom:* The units of measurement used by the dimension.

ODE::Dimension::Limit

Defines an interval of values by specifying an upper or lower bound (or both, if two **Limits** are used). The value field specifies the boundary. The **Operation** enumeration specifies the operator applied to the interval.

- *value*: the value which specifies the limit.
- *operation:* the type of operation used by the limit.

Examples:

- A Limit with operation "LessThanOrEqual" and value "5" would specify all values <= 5.

- Two Limits defined as {LessThanOrEqual, 10} and {GreaterThanOrEqual, 0} specifies a range between 0 and 10 inclusive.

ODE::Dimension::Operation

Specifies comparison operations to specify a range of values in the **Limit** element. These operations are LTE (<=), LT (<), GTE (>=), GT (>), and Equal (==).

ODE::Dimension::UnitsOfMeasurement

Enumeration of all supported units of measurements. These units are taken from the International System of Units but must not be limited to it. This enables also automated conversion between different units during matching.

ODE::Dimension::Binary

A **Binary** dimension (i.e., true/false, on/off etc).

ODE::Dimension::Categorical

A **Categorical** dimension matches on qualitative values. The covered **CategoryMembers** specify guaranteed or demanded qualitative values. The **SubsetType** defines how corresponding **Demands** and **Guarantees** are matched.

Example:

- ASIL, where the Demand is subset of the Guarantee:
    - Demand: Requires ASIL C
        - *type* = asil, *covered* = {A, B, C}
        - The system requires at least ASIL C. An ASIL C system would also give guarantees for ASIL A and B.
    - Guarantee: Provides ASIL D
        - *type* = asil, *covered* = {A, B, C, D}
        - An ASIL D level system provides ASIL A-D guarantees.
    - Demand is subset of Guarantee, therefore cooperation is possible.
- Colour, where the Guarantee is subset of the Demand:

- Demand: Requires blue or red colour
  - *type* = quality, *covered* = {blue, red}
- Guarantee: Provides red colour
  - *type* = quality, *covered* = {red}
- Cooperation is possible because red is one of the demanded colours.

CategoryMember

Specifies a qualitative value within a **Categorical** dimension.

3. SAFETY ANALYSIS MODELS

3.1 CONCEPT

Unlike the preceding packages, the ODE already has existing metamodels for modelling failure logic and safety analysis techniques. The ODE::FailureLogic package provides constructs for modelling failures along with their causes and probabilities, while the FTA, FMEA, and Markov sub-packages support those particular safety analyses.

However, these models were primarily envisioned as acting as evidence for safety argumentation, not as executable models that can be used directly at runtime. Furthermore, with the exception of the Markov package, their capabilities with respect to analysing dynamic systems are very limited.

To address these issues, the existing safety packages have been expanded and/or modified and a new package for Bayesian network analysis has been added to further support safety analysis of dynamic systems.

3.2 FAILURE LOGIC PACKAGE

The original ODE::FailureLogic package contained four main elements:

- **Failure**, which represents a failure of a component or system and includes a failure type and failure rate, as well as a flag signifying whether or not it is a common cause failure (CCF).
- **ProbabilityDistribution** and **ProbabilityDistributionParameter**, which are used together to describe the parameters for failure probability estimation and quantitative safety analysis.
- **MinimalCutSet** and **MinimalCutSets**, which represent the causes of a failure (in the form of more failures). This term originates from FTA, where root causes of a system-level failure are described as a disjunctive set of conjunctions of component-level failures.
- **FailureModel**, which represents an overall model of failure or safety analysis and encapsulates all of the Failures and their MinimalCutSets. It also serves as the parent class for all the various analysis technique sub-packages (so e.g. a FaultTree and a MarkovModel are both FailureModels).

Though all of these elements are retained in the new ODE version, some aspects have been modified to address the various issues identified. For example, the semantics around CCFs have been clarified and the Minimal Cut Sets extended to support sequences (and thus dynamic FTA) as well as non-failure causes. Furthermore, links to Events and Actions have been added and additional State-related fields have been added to link failures to system states (see also the next section on state machines).

The new FailureLogic metamodel is shown below:

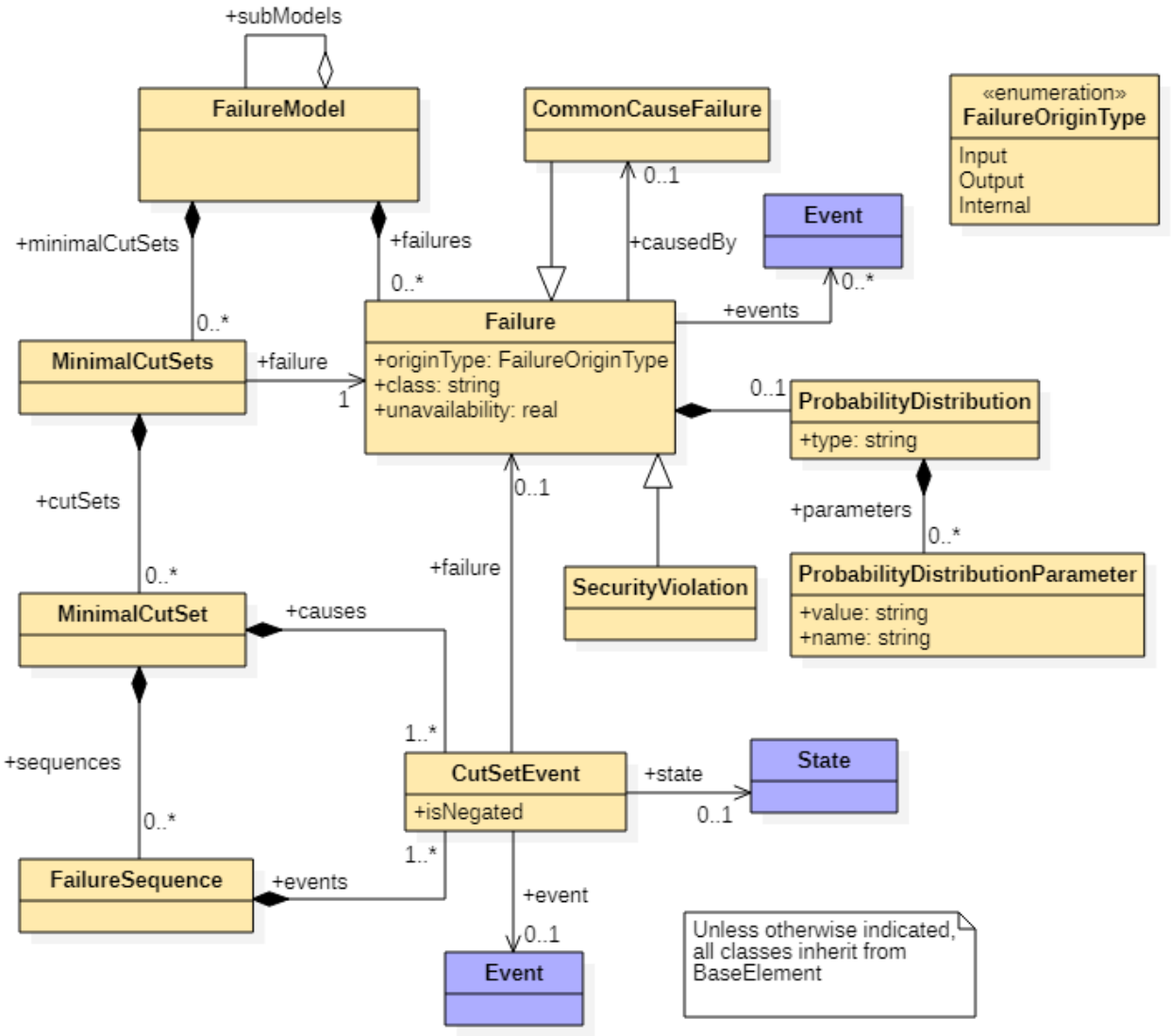


Figure 14 - ODE::FailureLogic metamodel

ODE::FailureLogic::FailureModel

Represents any form of failure model. Specialisations include fault trees, FMEAs, Bayesian networks, and Markov chains. The causes of failure(s) (whatever those might be) are represented as part of the failure model using minimal cut sets.

- *subModels[0..*]*: Allows decomposition of a failure model into sub-models.
- *minimalCutSets[0..*]:* Represent the causes of the analysed system/function/component failure(s) being modelled in the form of minimal cut sets.
- *failures[0..*]:* The failure(s) identified as part of the analysis model.

ODE::FailureLogic::Failure

The principle element of the package: represents a generic failure of a function/system/component. Failure is used for both root causes (e.g. component-level failures) and top-level system failures. Individual failure rates are modelled by the ProbabilityDistribution. A direct link to Events has also been added; this is so that there is a possible relation between the two that does not rely on another model (e.g. a fault tree) to act as an intermediary.

- *originType*: The origin of the failure relative to the parent system/function/component, as indicated by the **FailureOriginType** enumeration: Input (propagated to this element), Output (propagated from this element), Internal (originating and transformed inside this element).

- *class:* Describes the type/class of failure, e.g. 'Commission', 'Omission', 'Value' etc. Classes should be consistent across the model if propagation is to be modelled correctly.

- *unavailability:* If this is a fault tree top-event or system-level failure, then this field indicates the overall unavailability of the system (over the system lifespan) as a result of said failure. Note that this is typically derived from a fault tree analysis; the top event of a fault tree can itself be a failure, and since one of the outputs of the analysis is the unavailability of that top event, it is stored here for easier access rather than in the fault tree itself. For failures that are not top events, this field is ignored.

- *causedBy[0..1]*: The **CommonCauseFailure** that triggers this **Failure**, if any.

- *events[0..*]*: **Events** associated with this failure. At runtime, the conjunction of these events is interpreted as evidence that the failure has occurred. Conversely, if the **Failure** occurs, it is assumed the **Event** occurs too.

ODE::FailureLogic::CommonCauseFailure

A subclass of **Failure** that indicates a common cause failure (CCF), i.e., a failure that can trigger other dependent failures.

ODE::FailureLogic::MinimalCutSets

Represents the causes of the failures being analysed in the FailureModel.

- *cutSets[0..*]*: The minimal cut sets themselves.

- *failure:* The failure being caused by the cut sets (e.g. the top-event of a fault tree).

ODE::FailureLogic::MinimalCutSet
A conjunction/combination of causes that are necessary and sufficient to trigger a higher-level failure. Rather than contain failures directly, the minimal cut set now contains **CutSetEvents**, which encapsulate more information about each root cause of failure and which are not necessarily **Failures** themselves. There must be at least one **CutSetEvent**.

- *causes[1..*]:* The failure(s) or events(s) that cause the higher-level failure being analysed.

ODE::FailureLogic::FailureSequence
To model minimal cut sequences, a **MinimalCutSet** can contain one or more **FailureSequences**, which specify sequences across a subset of the events in the MCS. Again, this is in the form of **CutSetEvents**.

ODE::FailureLogic::CutSetEvent

Encapsulates more information about a failure: whether it is a complement event (i.e., negated; see *isNegated* flag below), the **State** (if any), and if it is a normal (non-failure) event, the associated **Event**.

- *failure:* the **Failure** being represented, i.e. the one that causes the *failure* element in the parent **MinimalCutSets** element. Either this or *event* must be set.

- *event:* the **Event** being represented, if the cause is a normal event and not a failure. Either this or *failure* must be set.

- *isNegated:* whether this event is negated (i.e., in a NOT gate). For example, a cut set may contain `HighPressure AND NOT ValveOpen` — i.e., two events in conjunction, (HighPressure) and (NOT ValveOpen) — in which case NOT ValveOpen is a "complement event" and would have this flag set. Given that a cut set contains all the failures/events necessary to cause some other failure/event, a complement event is so called because it must *not* occur for the other failure/event to occur.

- *state[0..1]*: the **State** the failure/event occurred in, if relevant. Generally speaking, two otherwise similar **CutSetEvents** with different states are treated as separate, though the exact behaviour is dependent on the analysis algorithm implementation.

ODE::FailureLogic::SecurityViolation

Represents the exploitation of a security weakness by an attacker and intended to create a link so that the impact of security violations can be modelled as part of a safety analysis.

ODE::FailureLogic::ProbabilityDistribution

Represents a type of probability distribution (e.g. exponential, Weibull, Poisson etc).

- *type*: Description of the type of distribution.
- *parameters[0..*]:* The parameters that define the distribution.

ODE::FailureLogic::ProbabilityDistributionParameter

Represents a parameter for a given probability distribution. The parameters used depend on the type of distribution, e.g. for exponential distributions, a failure rate, time, and optional repair rate are typically used.

- *name*: The name of the parameter (e.g. "Failure rate").
- *value:* The value of the parameter.

To expand further on the semantics of minimal cut sets and causes, consider a backup power supply system with the following causes of failure:

1. Battery failure when in backup mode (i.e., when the system is running off the backup supply);
2. Failure of mains power sensor, but only if the system has not already switched to backup mode (in which case the switch does not take place);
3. The system is switched off manually;
4. Overheating leading to a forced shutdown, e.g. due to prolonged usage in an environment with a high ambient temperature and minimal ventilation.

To model this with the ODE::FailureLogic package, we would have a **MinimalCutSets** element in which the system-level **Failure** under analysis is "No power from backup supply". The origin type would be "output" (since this is a failure of the output of the system) and the type would be some kind of omission failure (i.e., lack of power). The cut sets would then be as follows:

`MinimalCutSet #1: "Battery failure in backup mode"`

- `sequences: None`
- `causes:`
  - **`CutSetEvent`** `#1a`
    - `isNegated: false`
    - `failure:`
      - **`Failure`**

                - name: "Battery failure"
                - originType: internal
                - class: "component failure"
            - state:
                - **State**
                    - name: "On Battery Power"
            - event: None

MinimalCutSet #2: "Failure of mains power sensor"

- causes:
    - **CutSetEvent** #2a
        - isNegated: false
        - failure:
            - **Failure**
                - name: "Power Sensor failure"
                - originType: internal
                - class: "component failure"
        - state:
            - **State**
                - name: "On Mains Power"
        - event: None
    - **CutSetEvent** #2b
        - isNegated: false
        - failure: None
        - state:
            - **State**
                - name: "On Mains Power"
        - event:
            - **Event**
                - name: "Loss of mains power"
- sequences:
    - #2a before #2b

MinimalCutSet #3: "Manually switched off"

- sequences: None
- causes:
    - **CutSetEvent** #3a
        - isNegated: false
        - failure: None
        - state:
            - **State**
                - name: "On Battery Power"
        - event:
            - **Event**

- name: "Switched off"

MinimalCutSet #4: "Overheating"

- sequences: None
- causes:
  - **CutSetEvent** #4a
    - isNegated: false
    - failure: None
    - state:
      - **State**
        - name: "On Battery Power"
    - event:
      - **Event**
        - name: "Battery temperature exceeds safe threshold"

Here we can see how most features are being used. In MCS #1, the cause is a component failure: some failure of the battery itself. However, this only causes the system failure when the battery is in use, hence the State "On Battery Power" is defined.

For MCS #2, a sequence is needed. The system can only switch to battery power if it detects loss of main power; if the sensor fails before that point, no switch will take place. Conversely, if the sensor fails *after* the switch to battery power, it will not cause the failure.

For MCS #3, the cause is not a failure but rather a "normal event", i.e., an event that is expected as part of nominal system operation but may cause a problem in certain situations. In this case, the system is switched off when it is needed.

For MCS #4, the failure is caused by overheating, where the system shuts down to prevent a worse issue (e.g. fire). In this case, the cause is not a failure per se but rather an environmental condition. The source is therefore an Event (most likely originating from an onboard temperature sensor) rather than a Failure. Not that the failure of this temperature sensor could itself cause a different, more severe failure (e.g. the aforementioned fire).

3.3 STATE MACHINES & MARKOV MODELS

In the original ODE, the ODE::FailureLogic::Markov sub-package was primarily intended to support Markov chain analysis. However, when dealing with dynamic systems and runtime events, it can be useful to be able to model a more generic concept of state, whether for modelling dynamic failure logic or simply dynamic nominal behaviour.

To that end, the Markov package has been overhauled and expanded into a more generic ODE::FailureLogic::StateMachine package. While it continues to support Markov chains (now including transition tables), they are now handled as a probabilistic specialisation of a state machine. Moreover, it is now possible for transitions between

states to be deterministically triggered by Events and to trigger Actions in turn, linking the new StateMachines with other packages.

Since a StateMachine is still a FailureModel and there is already the capability for any system, function, or component to have a FailureModel, any system element can now also have a StateMachine to represent its dynamic behaviour.

The new StateMachine metamodel is displayed below:

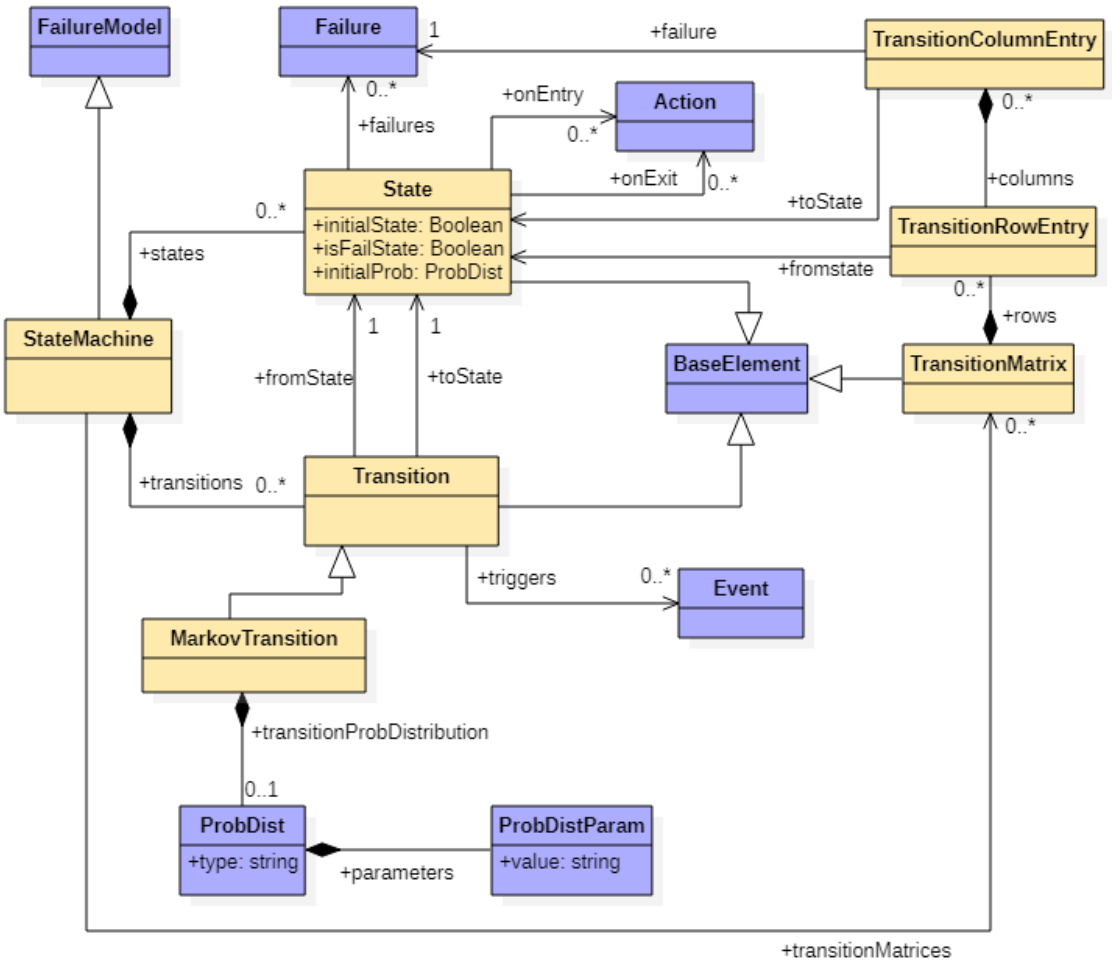


Figure 15 - StateMachine & Markov metamodel

ODE::StateMachine::StateMachine

This is the old MarkovChain renamed to a more generic form. It represents any type of state machine, whether probabilistic or deterministic.

- *transitions* [0..*]: The set of Transitions in the state machine/Markov chain.
- *states* [0..*]: The set of States in the state machine/Markov chain.

ODE::StateMachine::State
Represents a state of operation of a system/function/component. The system/function/component can only be in one state at a time, but sub-components could also have their own state machines etc. Generic state machines can be used to model nominal behaviour as well as failure behaviour, if required.

- *initialState:* A flag indicating whether this is the initial (starting) state. Note that more than one state can have this flag set, but in that case it should also have a probability of being the actual initial state (see *initialProb* below). This is to allow modelling of situations where the initial state is uncertain or otherwise non-deterministic.
- *initialProb:* The initial probability that this state is the starting state (for Markov models). If there is only one initial state, this would be 1.
- *isFailState*: A flag indicating whether this is a failure state.
- *failures[0..*]*: If this is a fail state, references the **Failure**(s) it represents.
- *onEntry[0..*]*: **Actions** to be undertaken when entering the state at runtime.
- *onExit[0..*]:* **Actions** to be undertaken when exiting the state at runtime.

ODE::StateMachine::Transition

Represents a generic transition from one state to another. Modified to ensure a 1:1 link between exactly one source state and one destination state and to allow triggering by **Events**.

- *fromState[1]*: The origin State.
- *toState[1]*: The destination State.
- *triggers [0..*]*: The **Event**(s) that trigger the transition. If more than one is referenced, a disjunction is assumed (i.e., any Event may trigger the transition).

ODE::StateMachine::MarkovTransition

The same as the old ODE::Markov::Transition: represents a probabilistic transition between states and may have parameters to describe its distribution.

- *transitionProbDistribution*: Describes the probability distribution of the transition (with appropriate parameters in the form of **ProbabilityDistributionParameters**).

ODE::StateMachine::TransitionMatrix

For more complex Markov models, a transition matrix can be used instead of the simpler *transitionProbDistribution* in **MarkovTransition**. This allows direct representation of each transition probability distribution within the Markov model itself, rather than each transition (as such, either one or the other approach should be used, not both).

- *rows*: The rows in the matrix.

ODE::StateMachine::TransitionRowEntry

A row in the **TransitionMatrix**. Linked to a given source state and displays (in each column) the transition probability to the given destination states.

- *columns*: The columns in the row.
- *fromState*: The source state from which the transition occurs.

ODE::StateMachine::TransitionColumnEntry

Represents the probability of transition to the given destination state.

- *failure*: The **Failure** that triggers the transition, including its probability distribution.
- *toState*: The destination state for this particular transition column.

The aim of these modifications is to allow generic state machines to model the dynamic behaviour of any given system element. Events and Actions are both runtime-relevant, meaning that a StateMachine provides a potential template for conversion to an EDDI that can respond to input (Events) and provide output (Actions).

Additionally, any FailureModel can have sub-FailureModels, meaning a top-level StateMachine could have other models (e.g. fault trees, Bayesian networks) to provide support for runtime fault diagnosis etc.

3.4 FAULT TREES

Fault Tree Analysis [4] or FTA is a widely used safety analysis technique that dates back to the 1960s and has been successfully applied across a variety of domains in the decades since. FTA is a top-down deductive approach, starting with a high-level failure and establishing its immediate causes, then repeating the process step-by-step for each of those causes and so on. It results in a logical structure relating a system failure (the "top event") to combinations of root causes ("basic events") via a network of Boolean logic gates. It supports both quantitative (probabilistic) and qualitative (logical) analysis

and has been extended in many forms over the years, some of which are described in D4.1.

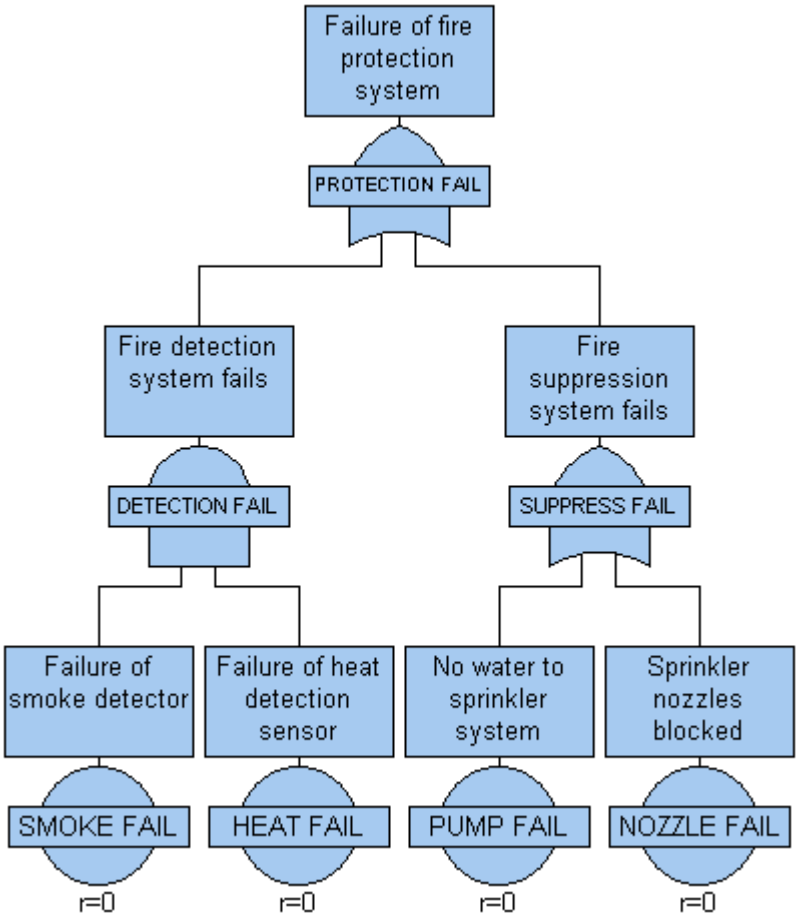


Figure 16 - Example fault tree

In the ODE, fault trees are modelled compositionally using **Causes**. There is a top-level container, the **FaultTree** itself, and then the various nodes of the fault tree are all modelled as Causes. This is because each node in the fault tree is the direct cause of the node above. To distinguish between intermediate causes that serve as logic gates, a subclass (**Gate**) exists.

The FaultTree package is very closely linked with the parent ODE::FailureLogic package, which defines failures and the FailureModel itself. The FaultTree package, at least in its original form, simply defined the compositional logic necessary to structure the tree. This generic compositional structure is broadly unchanged in the new version; however, new links have been added to other packages, particularly ODE::Event. The new metamodel is shown below.

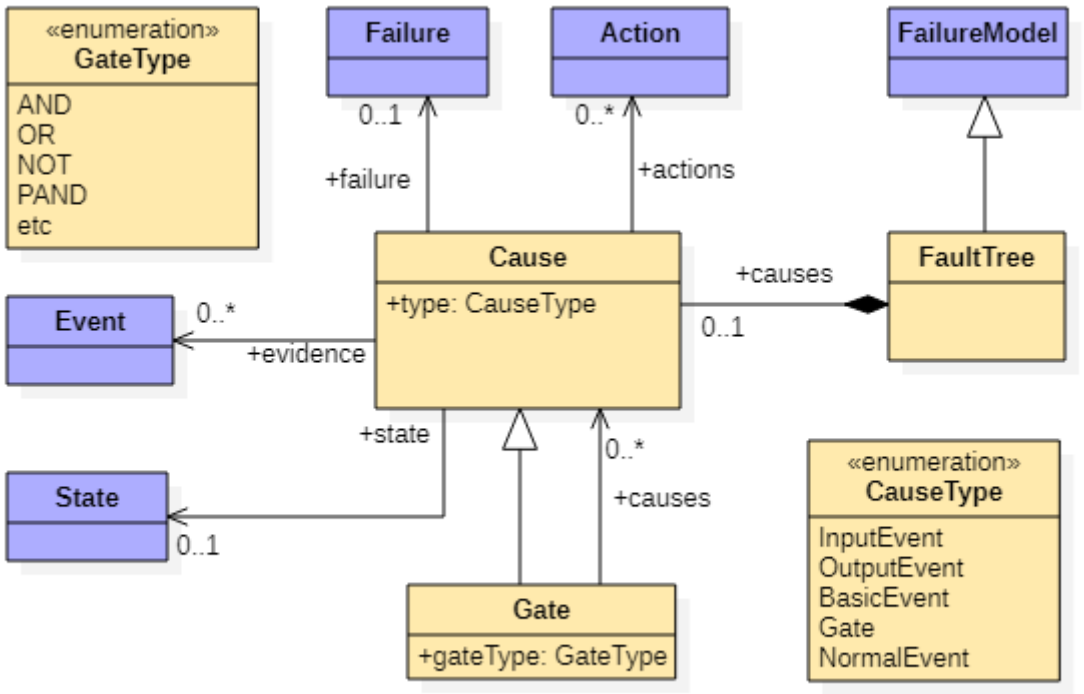


Figure 17 - FTA metamodel

ODE::FTA::FaultTree

Represents the results of a fault tree analysis in the form of a fault tree.

- *causes[0..1]*: The top event of the fault tree. For the fault tree to make sense, this **Cause** should have a top-level system **Failure** associated with it.

ODE::FTA::Cause

Represents a causal node in a fault tree. Causes can reference Events that provide evidence this Cause has occurred/is true; if all the events occur, the Cause is assumed to have occurred too. Causes can also reference a State, indicating that the Cause is only relevant in that particular state. If the state changes between a Cause and its parent, then it is assumed that the associated events/failures trigger a state change. Finally, Causes can now also link to Actions, which are triggered when the Cause becomes true.

- *actions[0..*]:* **Actions** triggered when the **Cause** has occurred/is true.

- *evidence[0..*]:* **Events** that indicate the cause is true and thus serve as evidence of its occurrence. Here the events are treated as a conjunction (i.e., the cause is only true if all events occur). This also serves as a mechanism to allow the fault tree to become executable, since nodes can become true as events occur and trigger actions in response.

- *failure[0..1]:* The concrete **Failure** (if any) associated with the cause. Primarily used for basic events, since most gates/intermediate nodes in the fault tree will not be associated directly with a single failure.

- *state[0..1]*: The relevant system state associated with this **Cause**.

- *type*: Whether the cause is an InputEvent, OutputEvent, BasicEvent, NormalEvent, or Gate (see corresponding enumeration). Input and Output Events correspond to Input and Output **FailureOriginTypes**, i.e. failures propagated to or from a component respectively. A BasicEvent corresponds largely to an Internal **FailureOriginType**, since it is generally a component failure originating within that component. NormalEvents are not failures at all but rather nominal events (and as such are expected to link to an **Event** rather than a **Failure**). Finally, **Gates** are logic gates and should only be used for the **Gate** subclass.

ODE::FTA::Gate

Represents a logic gate connecting one or more Causes and forms the tree structure itself via composition.

- *gateType*: The type of gate (see **GateType** enum). Boolean, non-coherent, and dynamic gates are included.

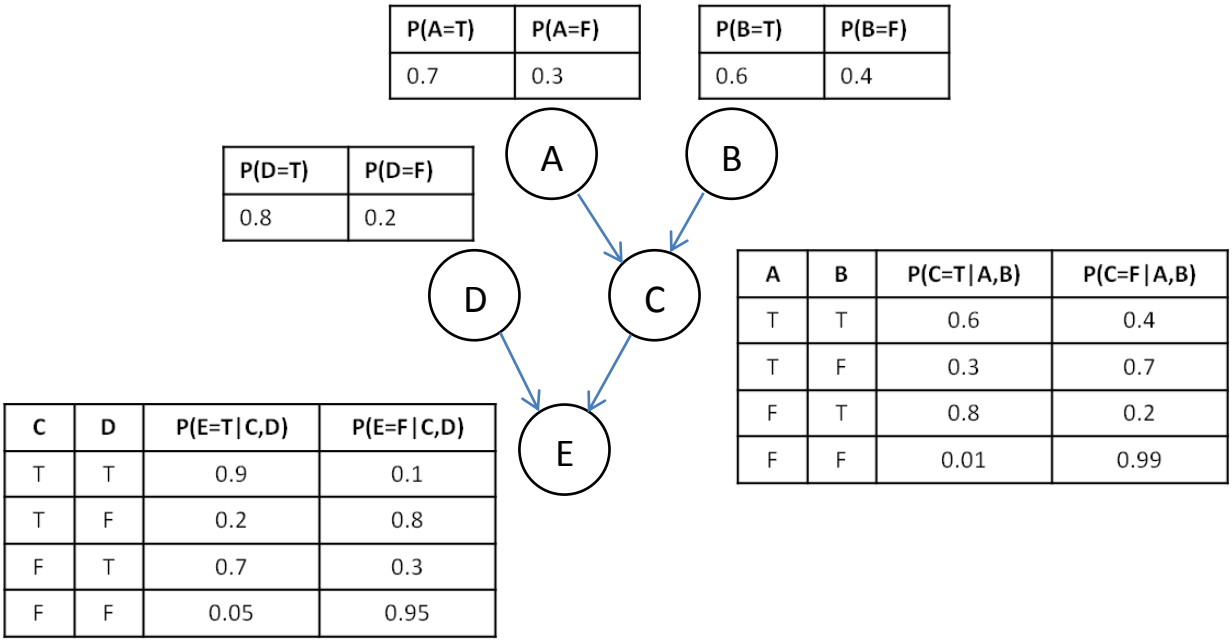
- *causes[0..*]:* The child causes that serve as input to this gate.

3.5 BAYESIAN NETWORKS

A Bayesian network (BN) is a graph structure that describes the conditional probabilities of the relationships between a set of random variables. While not specifically designed for use in dependability engineering, they have increasingly been used to model and analyse dynamic systems.

Bayesian networks have two main elements, nodes and edges, which can be used to represent the relationships between random variables. Unlike Markov chains, where the probability of the next state is memory-less and independent of any prior states, a Bayesian network operates on conditional probability and each edge indicates a conditional dependency. Conditional probability functions determine the status of each node given the status of the other nodes on which it is dependent. One of the advantages of Bayesian networks over similar approaches like Markov chains is that they are capable of inference and can satisfy probabilistic queries, e.g. inferring the knowledge about an unobservable node based on the status of the others that can be observed.

An example of a Bayesian network is given in Figure 18. The example network comprises five abstract random variables, i.e., the nodes. Each node has the states “T” (true) and “F” (false). The parametrisation is given in the tables close to each node consisting of the probability distributions given the parents.



P(A=T)	P(A=F)
0.7	0.3

P(B=T)	P(B=F)
0.6	0.4

P(D=T)	P(D=F)
0.8	0.2

A	B	P(C=T|A,B)	P(C=F|A,B)
T	T	0.6	0.4
T	F	0.3	0.7
F	T	0.8	0.2
F	F	0.01	0.99

C	D	P(E=T|C,D)	P(E=F|C,D)
T	T	0.9	0.1
T	F	0.2	0.8
F	T	0.7	0.3
F	F	0.05	0.95

Figure 18 - Example Bayesian Network (from [5])

The ODE::BayesianNetwork sub-package is a new addition to the ODE, added because of the expressiveness and ready application to dynamic systems. Its inference capabilities also make it particularly useful at runtime, where probabilities and states can be updated on the basis of real-time data.

The Bayesian network metamodel describes the elements required to formally describe and define an entire Bayesian network, as well as aspects required for the Dynamic Risk Assessment (DRA) runtime monitoring and potential analysis results. Some of the elements connect to other existing ODE package, including:

- the **ODE::FailureLogic** package (**FailureModel**)
- the **ODE::Event** package (**Event** and **Action**)
- the **ODE::Dimension** package (for **Dimension**)
- and the **ODE::SINADRA** package (various elements).

The respective metamodel and its connections are shown in below in Figure 19.

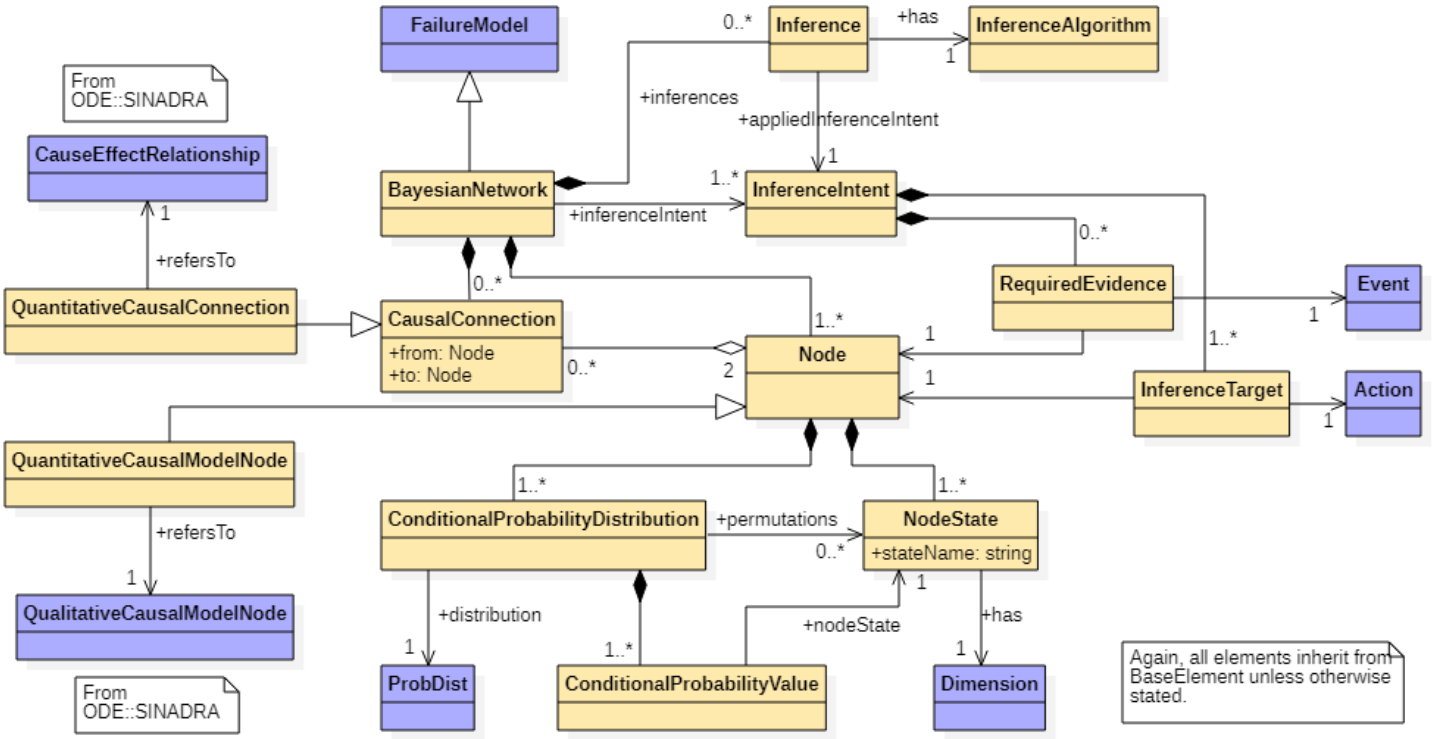


Figure 19 - ODE::BayesianNetwork metamodel

ODE::BayesianNetwork::BayesianNetwork

Element describing the whole Bayesian model (the entire directed acyclic graph) as a failure model. Thus, it inherits from **ODE::FailureLogic::FailureModel**. However, like a state machine, a Bayesian network can be used for several other purposes and intentions than just as a failure model, e.g., behaviour prediction for surrounding actors. The network comprises at least one **Node** and an arbitrary number of **CausalConnections** that link two nodes together each. A **BayesianNetwork** can have multiple **InferenceIntent**s depending on the semantics of the BN and the purpose of the engineer; each **InferenceIntent** specifies inputs and output **Nodes** of the network when used in practice. Further, it can have arbitrary many **Inference** objects associated.

- *nodes [1..*]:* The nodes in the network.
- *causalConnections[0..*]:* The 1:1 connections between the nodes.
- *inferenceIntents[1..*]:* The related **InferenceIntents**, as required to support the semantics.
- *inferences[0..*]:* The associated **Inferences**.

ODE::BayesianNetwork::Node

This element defines a node in the Bayesian model. It can describe arbitrary events and actions; hence, it is associated with **RequiredEvidence** and **InferenceTarget**. It can be connected to other **Nodes** via a **CausalConnection**. To describe the concrete discretization and parametrization of a node in a Bayesian network, at least one **NodeState** and one **ConditionalProbabilityDistribution** is required.

- *nodeStates[1..*]*: Describes the discretization of the underlying variable.
- *conditionalProbabilityDistributions[1..*]*: Describes the concrete parameterization of the variable given the parent **Nodes.**

ODE::BayesianNetwork::QuantitativeCausalModelNode

Element that describes the Bayesian network **Node** when the BN is derived from a SINADRA model. Hence, to assure traceability, the quantitative node (i.e., BN node) links to the corresponding qualitative causal model node (i.e., element in the situation-aware dynamic risk assessment model). The usage of this **QuantitativeCausalModelNode** is optional and only for the use case of deriving an inferable and quantified model from an actor behaviour cognitive model. Otherwise, the "normal" **Node** element should be used for Bayesian networks that unrelated to SINADRA.

ODE::BayesianNetwork::CausalConnection

Element that links two **Nodes** together. There is one parent and one child indicating the direction of the edge in the directed acyclic graph depicting the Bayesian network. This connection describes which **Nodes** influence which other **Nodes**. The **CausalConnection** should only be used for linking "normal" **Nodes** (and not SINADRA-related nodes).

- *from:* the source/parent node.
- *to:* the destination/child node.

ODE::BayesianNetwork::QuantitativeCausalConnection

Element that depicts a **CausalConnection** in the Bayesian network when the BN is derived from a SINADRA model. Hence, to assure traceability, the quantitative causal connection (i.e., BN edge) links to the corresponding cause effect relationship (i.e., element in the situation-aware dynamic risk assessment model). The usage of this **QuantitativeCausalConnection** is intended only for the use case of deriving an inferable and quantified model from an actor behaviour cognitive model, i.e., linking

two **QuantitativeCausalModelNode** together. Otherwise, the "normal" **CausalConnection** element should be used.

ODE::BayesianNetwork::NodeState

Element describing the states of a **Node**. Each **Node** has at least one **NodeState**. In that case, it is a constant state which would always be active because of missing alternatives. Typically, a **Node** would have multiple **NodeStates** that depict the different alternative states that the represented variable can be in.

- *dimension*[1]: For further defining the concrete discretization of the **NodeStates**, each **NodeState** has an **ODE::Dimension** which explicitly defines the discretization.

ODE::BayesianNetwork::ConditionalProbabilityDistribution

Element defining the concrete influences of parent nodes on a (child) **Node**. If a Node has no parents, the **ConditionalProbabilityDistribution** defines a probability distribution for the states of the **Node** without further dependencies. If a **Node** has at least one parent, it defines the probability distribution for the states of the **Node** given one permutation of the parents' states as a dependency each. Each permutation possible must be defined. The respective probability distribution is then defined using the **ODE::FailureLogic::ProbabilityDistribution** class.

- *distribution[1]*: The related probability distribution.
- *parentNodeStates[0..*]:* **NodeStates** describing the permutations of the node's parent states.
- *conditionalProbabilityValues[1..*]*: The conditional probability values for each **NodeState**.

ODE::BayesianNetwork::ConditionalProbabilityValue

Element that depicts the specific probability distribution value for a **NodeState** of a **Node** given a corresponding concrete permutation of states from the **Node**'s parents.

- *value*: The value
- *nodeState*: The relevant NodeState.

ODE::BayesianNetwork::InferenceIntent

Element that describes the interface of the **BayesianNetwork** itself. This includes the inputs, i.e., **RequiredEvidence**, and the outputs, i.e., **InferenceTarget**s, of the inference process. The described intent must contain at least one output to be useful.

- *requiredEvidence[0..*]:* inputs to the BN in the form of required evidence (provided by **Events**).
- *inferenceTargets[1..*]:* outputs of the BN in the form of targets (that translate to **Actions**).

ODE::BayesianNetwork::RequiredEvidence

Element that describes the inputs of the Bayesian network's inference process. It links to an **ODE::Event::Event**. Also, it links to a concrete node to which the input is mapped to.

- *node:* The concrete node.
- *event:* The input event.

ODE::BayesianNetwork::InferenceTarget

Element that describes the outputs, i.e., class nodes, of the Bayesian network's inference process. It links to an **ODE::Event::Action**. Also, it links to a concrete node which depicts the class nodes' inference targets.

- *action*: The corresponding action output.
- *node*: The concrete node.

ODE::BayesianNetwork::Inference

Element that depicts a concrete Bayesian network inference. That includes an associated **InferenceAlgorithm** that should be used to infer the output nodes. Also, it is linked to a concrete **InferenceIntent** that is used for that specific **Inference** and defines the specific input- and output-interface of the inference. This is important because a BN can have multiple **InferenceIntent**s which have different sets of input and output and thus evidence and class nodes.

- *inferenceAlgorithm:* The associated inference algorithm.
- *inferenceIntent:* The related **InferenceIntent**.

ODE::BayesianNetwork::InferenceAlgorithm

Element describing the concrete algorithm that should perform the **Inference** step, i.e., the computation of the not yet assigned nodes given the assigned nodes (evidences). The algorithm could be either stochastic (approximate) or exact. There is a plethora of different algorithms in both categories, each varying in accuracy and performance.

3.6 FMEA

An FMEA (Failure Modes & Effects Analysis) is an inductive safety analysis technique. The goal of an FMEA is to review the components of a system, identifying the potential failures of each one and then assessing the potential effects of those failures. Dating back to the late 1940s [6], it is one of the first structured safety analysis techniques. While it can be effective in establishing the risk posed by low-level failures — based on severity, probability, and detectability — it has a number of drawbacks. Foremost amongst these is the inability to consider the effects of combinations of failure modes, which would rapidly spiral out of control if even pairs of failures are considered, let alone other combinations.

Table 3 - Example FMEA table

Item	Failure Mode	Effect	Potential Causes	Sev.	Prob.	Det.	RPN	Action
Smoke detector	Does not detect smoke	Failure to detect fire	Sensor malfunction	8	3	6	144	Replace detectors on a regular basis
			Battery failure		7	3	168	Ensure regular battery testing
Heat sensor	Does not detect heat	Failure to detect fire	Sensor malfunction	8	3	4	96	Replace detectors on a regular basis
			Gremlins		1	8	64	Do not feed Mogwai after midnight
Sprinkler	No water	Failure to extinguish fire	Water supply disruption	6	3	4	72	Add local water storage for emergency use
	Nozzle blockage	Failure to extinguish fire	Dust/debris infiltration	6	2	5	60	Ensure nozzles are cleaned regularly
Alarm	No sound	Failure to warn of fire	Speaker malfunction	7	2	7	98	Test alarm periodically

While unchanged from its original form, the FMEA sub-package is described here for completeness. Note that the ODE::FMEA supports FMEDA (Failure Modes, Effects, & Diagnostic Analysis), which adds more quantitative failure data and detectability metrics.

The metamodel is shown below.

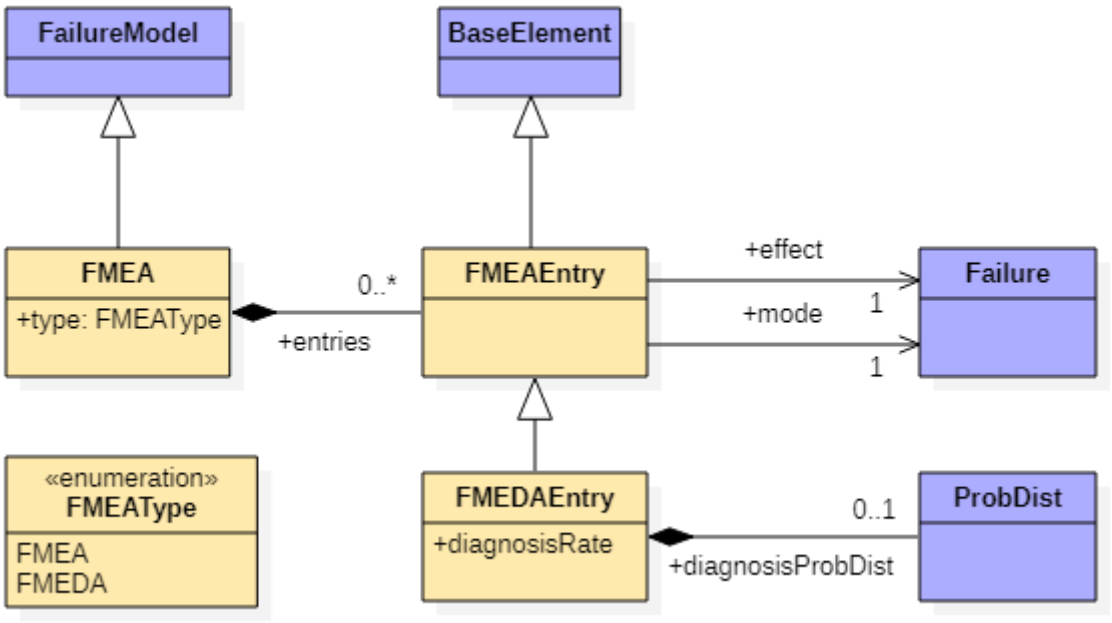


Figure 20 - ODE::FailureLogic::FMEA metamodel

ODE::FMEA::FMEA

The parent FMEA container. Contains all elements relevant to an FME(D)A.

- *type*: Whether this is a normal FMEA or an FMEDA.
- *entries[0..*]:* The entries (or rows) of the FMEA table.

ODE::FMEA::FMEAEntry

An entry in an FMEA table, containing a failure mode and its effects.

- *effect*: The effect caused by the **Failure** (in the form of a higher-level failure).
- *mode:* The **Failure** that causes the effect (typically a lower-level component failure mode).

ODE::FMEA::FMEDAEntry

In the case of an FMEDA, this element is used instead. It includes the diagnosis rate as a measure of detectability and a related diagnosis probability distribution.

- *diagnosisRate*: The rate of diagnosis over the assumed time period, e.g. mission time.
- *diagnosisProbabilityDistribution:* The related probabilistic distribution.

4. SECURITY ANALYSIS MODELLING

In this section we describe the need for additional classes in different packages of the ODE metamodel to capture the information that is delivered from the SESAME security assessment process. SESAME security assessment follows a structured series of steps, which overlaps in many cases with the highly structured process of threat modelling and its clearly specified steps, based on the chosen model. The aforementioned steps are described in details in **D5.1 Security Analysis Concept & Methodology for EDDI development**. The output of the SESAME security assessment is potential attack scenarios, in the form of a graph (attack trees) and accompanying mitigations, derived from the identified known vulnerabilities of a given system.

What follows is the description of the proposed classes per package. An example security analysis can be found in **D4.3: Safety/Security Co-Analysis Framework**.

4.1 FAILURELOGIC AND FTA PACKAGES

The existing Fault Tree Analysis (FTA) package can be used for the description of the potential attack tree graphs, one of the outcomes of the security assessment. As described in the previous section, fault trees already provide a hierarchical tree data structure that can be reused

The existing entities of this package are sufficient for modelling fault trees and corresponding attack trees. Attack trees are hierarchical diagrams that show how hostile activities can be combined to achieve an attacker's objectives. SESAME security assessment identifies possible attacks regarding a given system and delivers combinations of these attacks in the form of attack trees.

Classes from the existed FailureLogic package can be also used in combination with the FTA package. According to the ODE metamodel, a fault tree consists of generic causes, and each cause is possibly associated with a failure of a function/system/component. Moreover, a generic failure might be caused by a security violation. This is represented by the SecurityViolation, a subclass of Failure, and part of the FailureLogic package.

According to the proposed extension of the ODE metamodel SecurityViolation class can be used as a superclass for the CommonAttack class. The CommonAttack class is the equivalent of the Attack class that already exists in the Threat Analysis and Risk Assessment (TARA) package. In that way, FailureLogic and FTA packages are connected with the proposed classes of the TARA package and all together are able to capture the information regarding a graph of potential attack scenarios, output of the SESAME security assessment process.

Figure 21 below depicts the proposed classes for TARA package and their relationships with the corresponding classes of the FailureLogic and FTA packages. The proposed classes, part of the TARA package, are described in the next subsection. The rest of the classes in TARA could be also used for additional information if it is available.

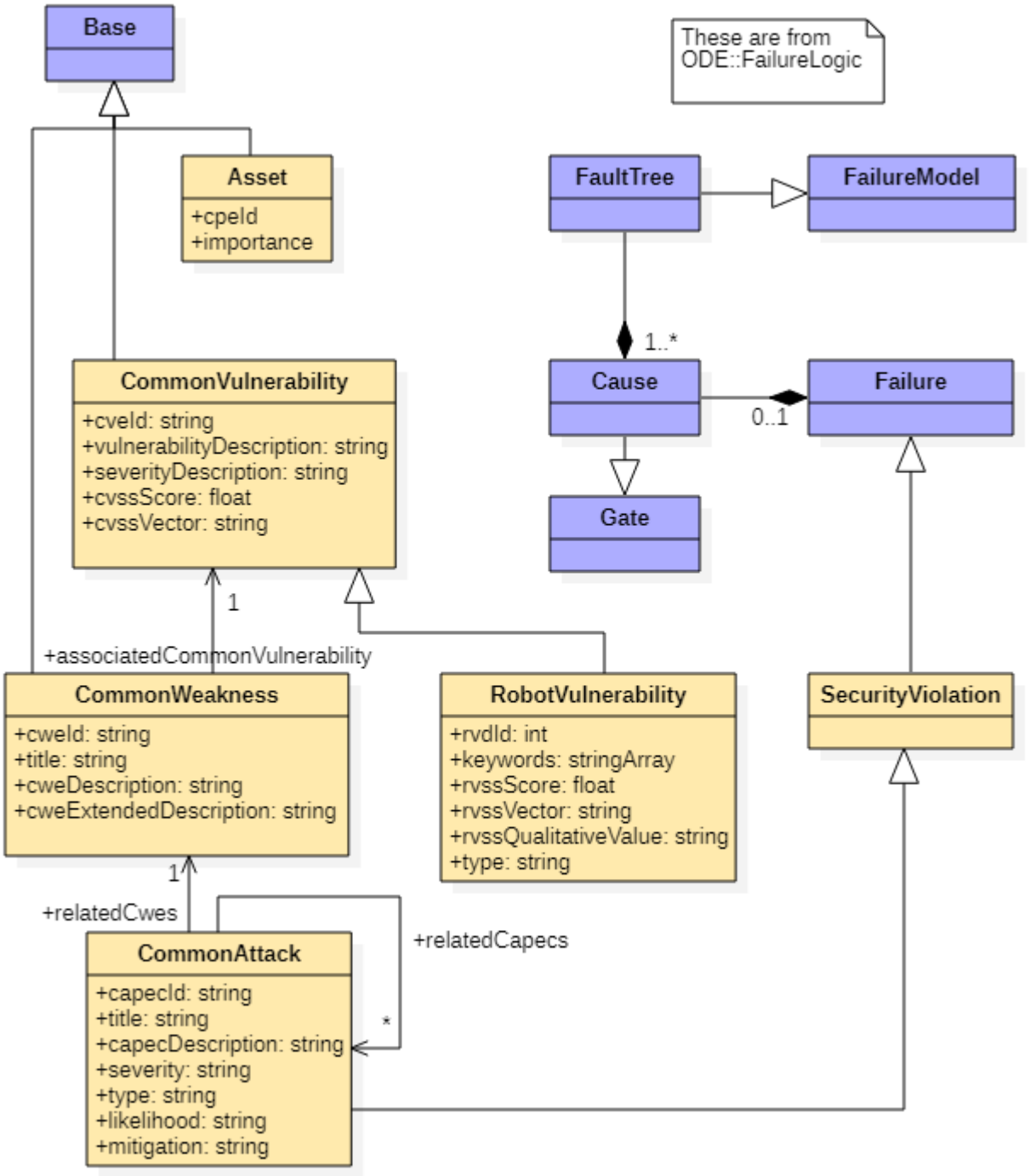


Figure 21 - proposed additions for the TARA package among with their relationships with classes of the FailureLogic and FTA packages

4.2 EXTENSIONS TO ODE::DEPENDABILITY::TARA PACKAGE

Classes included in the TARA package of the ODE metamodel can be used for defining the security status of a given system, the outcome of the security assessment process. However, there is a need for additional classes and fields in some of the existing classes to store all the information created.

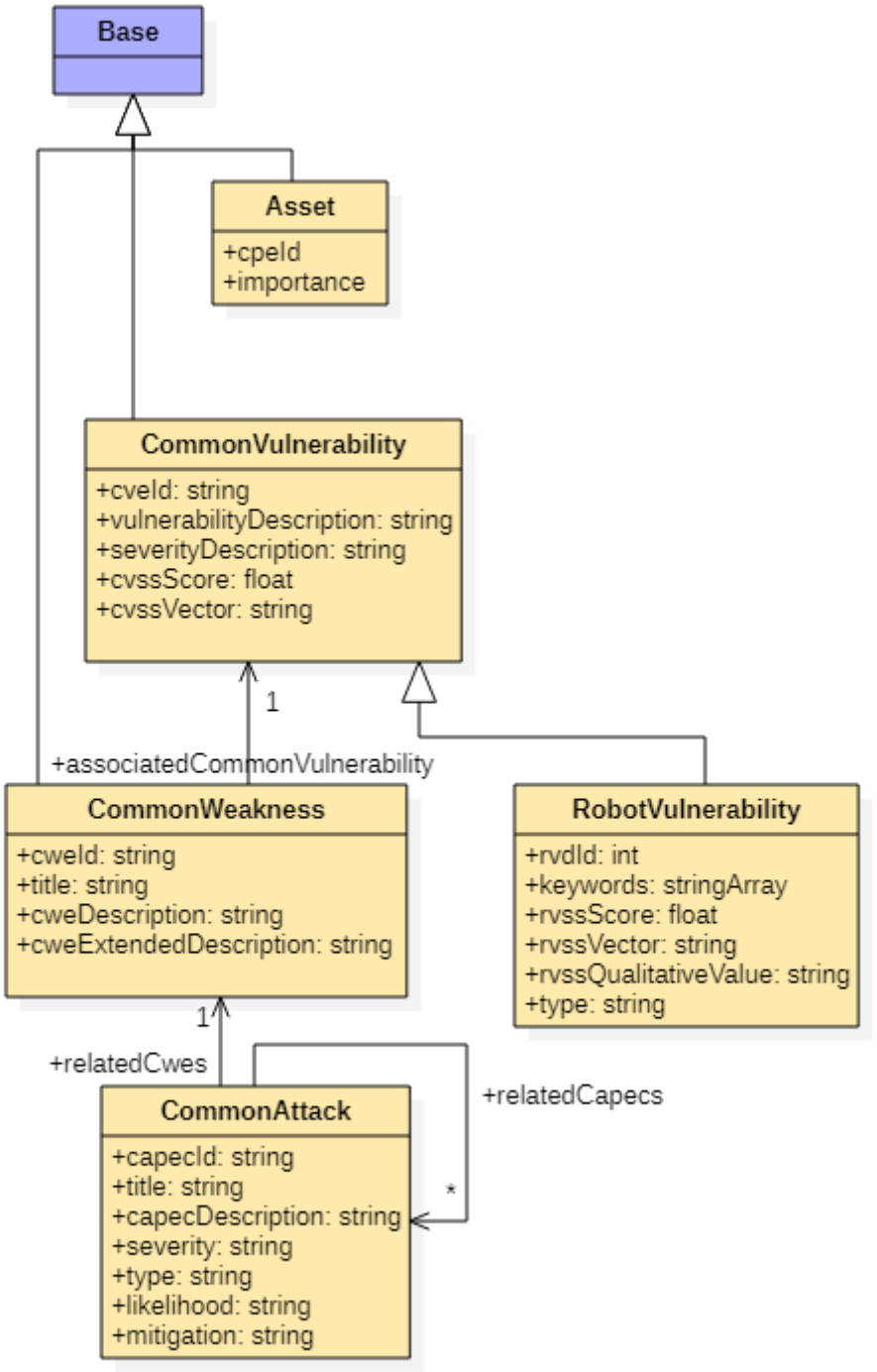


Figure 22 - proposed additions for the TARA package

ODE::Dependability::TARA::CommonAttack

Represents a known attack that can be conducted by taking advantage of the weaknesses and vulnerabilities of a system.

- *capecId*: a unique integer identifier that the CAPEC repository assigns to each known attack. Outside the CAPEC repository, attack patterns are referenced using this identifier in the form "CAPEC-####".

- *title*: the title of the attack as it is mentioned in the CAPEC repository. The purpose of this field is to be descriptive but short at the same time.

- *capecDescription:* a description of the attack including potential information about the context and the potential attacks that could take advantage of the weakness.

- *severity*: a qualitative value that describes the severity of the attack. Since the severity of a specific attack instance can vary greatly depending on the specific context of the target software, this field delivers an overall typical average value for this type of attack. The scale includes the values Very Low, Low, Medium, High, and Very high.

- *type*: represents that type of attack as it is listed in the CAPEC repository. The values can be Meta attack pattern, Standard attack pattern, Detailed attack pattern.

- *likelihood*: the probability an attack can be conducted taking into consideration attack prerequisites, the attack surface of the targeted system, skill and resources required, and available solutions. Once again, this field delivers an overall typical average value for this type of attack, since the likelihood of exploit of a specific attack instance can vary greatly depending on the specific context of the target software. The scale includes the values Very Low, Low, Medium, High, and Very high.

- *mitigation*: mitigation actions that could make the attack ineffective. The described mitigations could i) improve the resistance of the target software, ii) reduce the likelihood of the attack's success, iii) improve the resilience of the target software. As a result impact of the attack can be reduced, if it is successful.

- *relatedCapecs[1..*]*: a list of known attacks that are somehow related to the attack in question. The nature of the relationship between the attacks can be one of the following: i) Parent of, ii) Child of, iii) canFolllow, and iv) can Precede. Information regarding attacks that can potentially follow or precede the attack in question will be very useful for the creation of the attack trees in describing a whole attack scenario.

- *relatedCwes[1…*]*: a list of known weaknesses that are associated with the specified CAPEC. The mentioned weaknesses reference the industry-standard Common Weakness Enumeration (CWE).

ODE::Dependability::TARA::CommonWeakness

Represents a software and hardware weakness type. It serves as a baseline for weakness identification, mitigation, and prevention efforts.

- *cweId*: a unique identifier that the CWE repository assigns to each software and hardware weakness. Outside the CWE repository, weaknesses are referenced using this identifier in the form "CWE-####".
- *title*: the title of the weakness as it is mentioned in the CWE repository. This field is a short text description of the weakness allowing each weakness to be clearly identified.
- *cweDescription:* a text description of the weakness in question with a little bit more information than the title of the weakness.
- *mitigations*: an extended description of mitigation actions that can be used for the mentioned weakness to be avoided or to make the weakness easier to be avoided.
- associatedVulnerability: the vulnerability that is associated with the CWE.

ODE::Dependability::TARA::CommonVulnerability

Represents a computer security flaw, cybersecurity vulnerability, listed in the CVE security knowledge repository.

- *cveId*: a unique identifier that the CVE repository assigns to each vulnerability. The form of the CVE ID is "CVE-year-id number". An id number is a sequence number of four or more digits.
- *vulnerabilityDescription*: a description of the vulnerability itself, including information about the affected software/hardware and the potential types of attacks. Additional information that the description may include is vulnerabilities that are somehow related to the vulnerability in question.
- *severityDescription*: a description of the severity of the vulnerability including the base score and vector.
- *cvssScore*: the Common Vulnerability Scoring System numerical score reflecting the vulnerability severity. This field is a float that is the result of different metric groups that are taken under consideration: Base, Temporal, and Environmental.
- *cvssVector*: CvssVector is a text representation of a set of CVSS metrics. It is a form that is commonly used to record or transfer CVSS metrics. In this way, the information is stored or transferred in a concise form.
- *vulnerableAsset*: the asset that is affected by the vulnerability.

ODE::Dependability::TARA::RobotVulnerability

Represents a computer security flaw, cybersecurity vulnerability, listed in the Robot Vulnerability Database (RVD) security knowledge repository. It is a subclass of the Vulnerability class including some attributes that are unique to the RVD repository.

- *rvdId*: a unique identifier that the RVD repository assigns to each vulnerability. Outside the RVD repository, vulnerabilities are referenced using this identifier in the form "RVD#sequence_number".

- *keywords*: a set of strings that describe a vulnerability, helping its classification. These keywords may include information about i) robot and robot components that the vulnerability in question is present, ii) the vendor, iii) type (vulnerability, bug, …), iv) severity, v) version, and vi) if the vulnerability can be mitigated.

- *rvssScore*: Robot Vulnerability Scoring System (RVSS) takes under consideration a) robot safety aspects, b) assessment of downstream implications of a given vulnerability, c) library and third-party scoring assessments and d) environmental variables.

- *rvssVector:* RvssVector is a text representation of a set of RVSS metrics. It is a form that is commonly used to record or transfer RVSS metrics. In this way, the information is stored or transferred in a concise form. The values of this field are similar to the cvss-vector.

- *rvssQualitetiveValue:* this field includes a qualitative textual representation of the rvss score. The possible values can be None, Low, Medium, High, Critical.

- *type*: a classification of the RVD entries including vulnerabilities, bugs and others.

ODE::Dependability::TARA::Asset

Represents a concept/entity related to the operation of the system that is valuable for its stakeholders. This class is already present in the current version of ODE. The already existing fields are the following:

- *financialValue:* Represents the financial cost of acquiring/owning the Asset; can be interpreted as the cost of loss, replacement, etc.

- *financialCostType:* The monetary unit financialValue is measured in e.g. US dollars, Euro, etc.

- *operationalCost:* Represents the cost in time of the Asset being unavailable with respect to the system's operation; can be interpreted as the cost of operation halt, etc.

- *operationalCostType: The time unit in lost workhours, days, or months.*

However, a set of additional field are necessary for capturing the information that is required for i) the identification of known vulnerabilities associated with a given asset and ii) the calculation of the overall security score of the system in question.

- *cpeId*: a unique identifier for platform types. CpeId follows the cpe: <cpe_version>:<part>:<vendor>:<product>:<version>:<update>:<edition>:<language>:<sw_edition>:<target_sw>:<target_hw>:<other> format, maintained by NIST[5]. For example, "cpe:/o:microsoft:windows_xp:::pro" stands for Microsoft Windows XP Professional.

- *importance*: represents the role of an asset in the overall functionality of the system in question, determining how the corresponding vulnerability will be confronted. It could be a function of fields already present in the Asset class such as *financialValue* and *operationCost,* if such fields can be specified by the system administrator.

4.3 EXTENSIONS TO ODE::EVENT PACKAGE

As described earlier, there are four general categories of Event: i) "Condition" or "observed" events, which are conditions that hold over one or more variables/values and which occur when those conditions are observed to be true; ii) "Intelligent" or "Machine Learning" events, which are triggered as a result of some kind of decision-making process and which may involve some degree of uncertainty; iii) "Security intrusion alerts" events, triggered from an attack detection that is made by an Intrusion Detection System (IDS), and iv) "External" events, which are received externally from other EDDIs operating in the wider context, including information such as changing safety guarantees of a robot of an MRS. In this section, we focus on the IDSEvent and IDSEventMonitor classes that are closely related to the proposed security extensions for the ODE metamodel.

[5] https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir7695.pdf

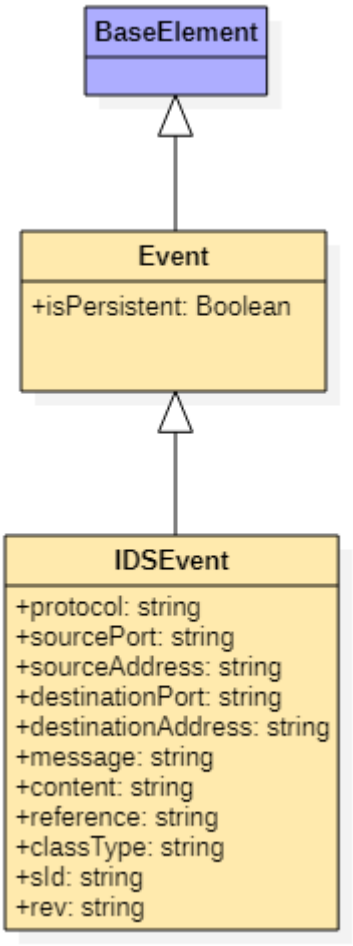


Figure 23 - IDSEvent proposed class

ODE::Event::IDSEvent

Represents a complex event triggered by a detected security intrusion. It is a subclass of the Event class. The fields provided are expected to be populated from the log files of the IDS, although the final implementation may only require a subset, depending on the system and IDS in question.

- *protocol:* the protocol that is used for the transmission of the suspicious packet (TCP, UDP, ICMP, and IP)
- sourceAddress: the source IP address of the suspicious packet
- sourcePort: the source port of the suspicious packet. It may be specified in a number of ways, including any ports, static port definitions, ranges, and by negation
- destinationAddress: the destination IP address of the suspicious packet
- destinationPort: the destination port of the suspicious packet.
- message: the message to be included at a created alert of the IDS. It is a meaningful message that includes what the rule is detecting. It's format is a simple text string.

- content: specific content in the suspicious packet payload that triggers an alert/action. This is an important feature that allows for the trigger of a response based on that data. The option data can contain mixed text and binary data.

- reference: a reference to external attack identification systems. It is a piece of additional information about the alert produced

- classType: used to categorize a rule as detecting an attack that is part of a more general type of attack class. Examples could include “trojan-activity”, “attempted-dos”, and “suspicious-login”

- sId: a unique identifier of Snort rules

- rev: an identifier of revisions of a rule

ODE::Event::IDSEventMonitor

Represents information about how an event will be monitored/detected, including implementation specific details. It is a subclass of the EventMonitor class. The details here are still to be determined but will be based on the requirements for a generic runtime intrusion detection system.

5. DYNAMIC RISK ANALYSIS

5.1 CONCEPT

Runtime safety approaches such as the analysis techniques in Section 3 generally focus on detecting faults and diagnosing their causes for safety-related errors or failures within the multi-agent system (or its constituent platforms). However, whether a particular error or failure is safety-critical and poses an actual risk depends on the current operational situation the system finds itself in during runtime. For instance, if a planned trajectory of a drone differs from specification due to a fault in the system, a collision with other dynamic or static objects may only occur if those objects are present in the current operational situation. Thus, hazardous events and their associated risk are always conditioned on the operational situation.

Dynamic risk assessment (DRA) techniques thus treat the MAS or a constituent system as a black box and provide means to analyse the consequences of MAS behaviour deviations on the risk in the current operational situation. Note that DRA can be both performed for constituent systems and on the MAS as a whole. The difference lies in the definition of hazards resulting from deviating behaviour. Such hazards can be analysed for MAS collaborative behaviour or single system behaviour.

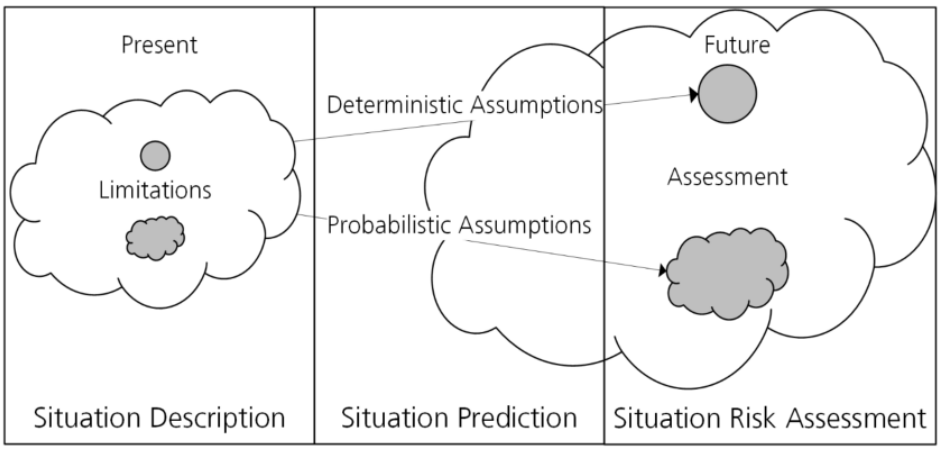


Figure 24 - Dynamic Risk Assessment Conceptual Overview [7]

Three main sub-tasks are required as part of DRA, as illustrated in the figure above:

1. **Situation Description**, which consists of a risk-driven decomposition of the situation space. Although this is similar to the worst-case assessment typically performed as part of a HARA, in a DRA, the risk is dependent on operational situations and thus extends the purely design-time analysis of HARA with a more detailed runtime-oriented analysis, e.g. one that involves environmental conditions as well as both dynamic and static objects and their potential interactions.

2. **Situation Prediction**, which involves predicting the future state of the relevant situation features based on assumptions and predictive models such as kinematics.

3. **Situation Risk Assessment.** Having predicted the future state of the situation, their relation to risk needs to be assessed according to relevant risk metrics.

To account for the fact that risk is often influenced by such situation-specific features, the *Situation-Aware Dynamic Risk Assessment* (SINADRA) framework [8] has been developed. The approach uses Bayesian networks as a modelling formalism and explicitly considers the mentioned situation-specific features as risk influences. SINADRA enables a system or MAS to automatically assess the risk of a (malfunctioning) behaviour in the *current* operational situation. Based on this dynamic risk estimate, the unavailability of safety capabilities can be potentially tolerated in low-risk situations or tactical decisions can be enacted to actively lower the risk to an acceptable level.

5.2 BEHAVIOUR MODELLING

This section describes the situation-aware dynamic risk assessment (SINADRA) metamodel. The intention of this model is to provide the required elements required to formally describe and define variations on the behaviour of controllable (e.g. autonomous robots) and uncontrollable actors (e.g. humans) based on the presence or absence of situation features.

The conceptual backbone of the metamodel is a decomposition of *behaviours* into *cognitive steps*, which enable actors to enact a behavioural decision based on *capabilities* for situation perception, reasoning and execution. The general idea is that there is a sequence of cognitive steps an actor follows to safely and efficiently accomplish a specific behaviour or sequence of behaviours (i.e., work tasks). Popular cognition models usable for both humans and machines are Sense-Plan-Act or, more fine-grained, Sense-Understand-Decide-Act. Risk can be associated with the likelihood and consequences of a particularly required capability being impaired or not present, represented by the *capability deviation*. A capability deviation is influenced by the presence or absence of situation features. Usually, situation features can be grouped, as they contribute to similar abstract *deviation influence concepts*, which have similar effects on a particular capability deviation.

Based on the aforementioned entities, we derived a proposal for the SINADRA metamodel, which is located in the **ODE::SINADRA** package and is depicted in Figure 25 below. Some of the model elements connect to other ODE packages like the existing **ODE::HARA** or the also newly proposed **ODE::BayesianNetwork** package. For the sake of readability, the inheritance connection for all the elements that are not indirectly inheriting from **ODE::ODEBase::BaseElement** are omitted.

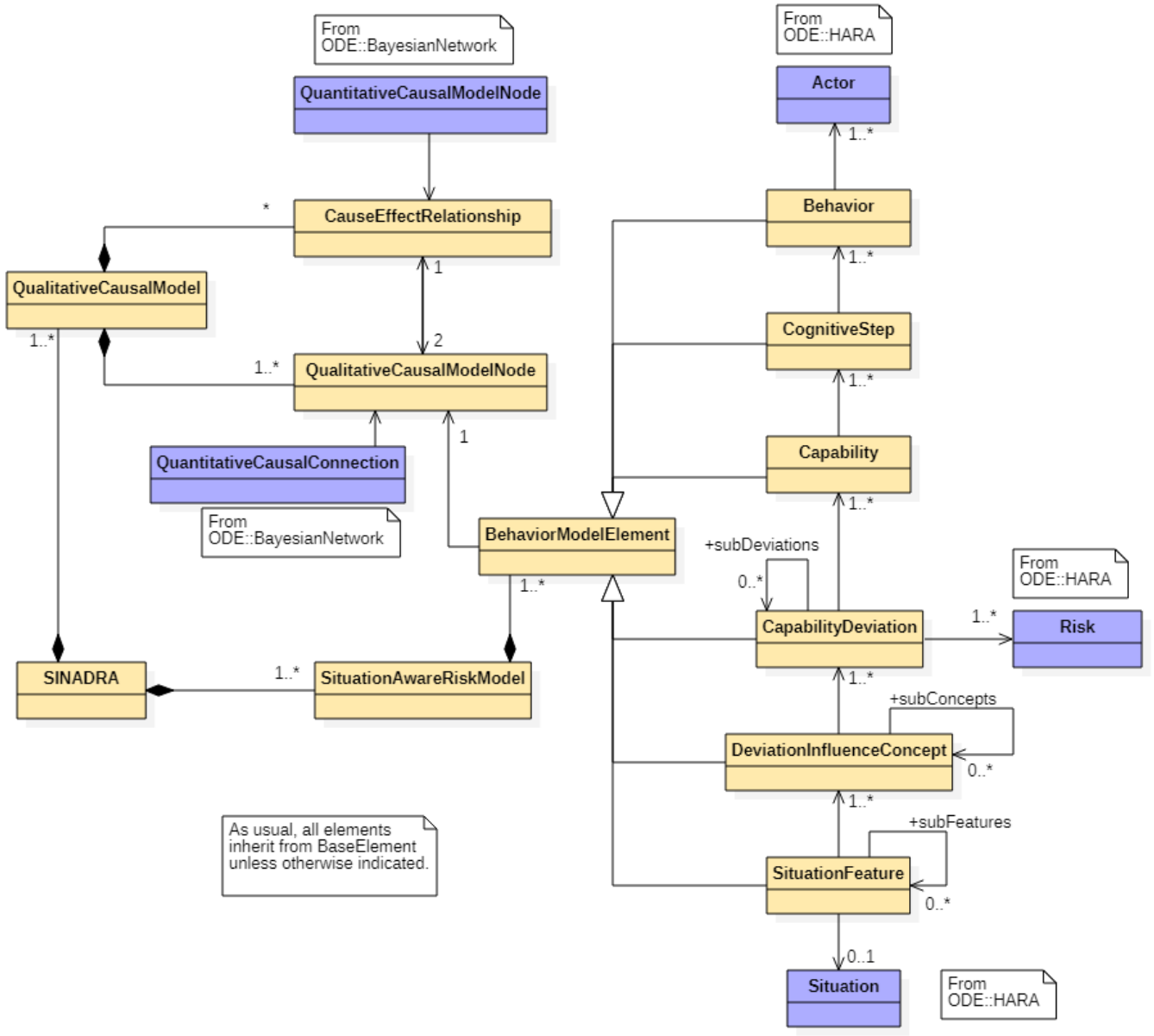


Figure 25 - ODE::SINADRA metamodel for situation-aware dynamic risk assessment

ODE::SINADRA::BehaviorModelElement

An abstract element from which all conceptual model elements (**Behavior**, **CognitiveStep**, **CapabilityDeviation**, **Behavior**, **DeviationInfluence-Concept**, **SituationFeature**) inherit.

- *causalModelNode:* The qualitative causal model node associated with this element.

ODE::SINADRA::Behavior

Element describing a behavior (i.e., a task, such as "move safely through corridor according to plan") of an (un)controllable actor. A behavior is linked to an **ODE::HARA::Actor**.

ODE::SINADRA::CognitiveStep
Element describing actor-related steps like “sensing”, “planning” or “acting”. At least one **CognitiveStep** contributes to enabling a **Behavior**.

ODE::SINADRA::Capability

Element describing actor-related capabilities (e.g. “paying attention to the scene”, “localizing other actors”) that contribute to achieving one or more **CognitiveSteps**.

ODE::SINADRA::CapabilityDeviation

Element describing an impaired or non-present capability (e.g. “paying no attention to the scene”, “existing actors are not localized”) which is influenced by the presence or absence of situation features. At least one **CapabilityDeviation** is associated with a **Capability**. A **CapabilityDeviation** can be linked to other **CapabilityDeviations**.

- *risk:* As a **CapabilityDeviation** can pose a risk in a specific situation, it is associated with **ODE::HARA::Risk**.

ODE::SINADRA::DeviationInfluenceConcept

Element describing an abstract deviation influence concept (e.g. “cognitive capacity”, “occlusion”) that emerges due to the presence or non-presence of at least one **SituationFeature**. A **DeviationInfluenceConcept** can be associated with other **DeviationInfluenceConcepts**.

ODE::SINADRA::SituationFeature

Element describing a feature which can be present or non-present (e.g. “lighting condition”, “unobservable corner”). It is associated with at least one **DeviationInfluenceConcept**. At least one **SituationFeature** is associated with an **ODE::HARA::Situation**. A **SituationFeature** can be associated with other **SituationFeatures**.

ODE::SINADRA::SINADRA

Overall container SINADRA element comprising at least one **SituationAwareRiskModel** and the associated **QualitativeCausalModels**.

- *causalModels [1..*]:* the causal models that form part of the SINADRA.

- *riskModels [1..*]*: the risk models that form part of the SINADRA.

ODE::SINADRA::SituationAwareRiskModel

Element that composes the elements of the conceptual model via the abstract **BehaviorModelElement**.

- *modelElements[1..*]:* the elements forming the behavioural risk model.

ODE::SINADRA::QualitativeCausalModel

Element that composes at least one **QualitativeCausalModelNode**. As there could be **QualitativeCausalModels** consisting of only one **QualitativeCausalModelNode**, a **QualitativeCausalModel** can be composed of zero or more **CausalEffectRelationships**.

ODE::SINADRA::QualitativeCausalModelNode

Element that represents a qualitative causal model node that is associated with a **BehaviorModelElement**. Two **QualitativeCausalModelNodes** can be in a cause-effect relationship, which can be expressed by an association to a **CauseEffectRelationship** element.

ODE::SINADRA::CauseEffectRelationship

Element describing the cause-effect relationship of two **QualitativeCausalModelNodes**.

To illustrate a possible relation of behavior model elements from the **ODE::SINADRA** package (Figure 25), an example is depicted in Figure 26.

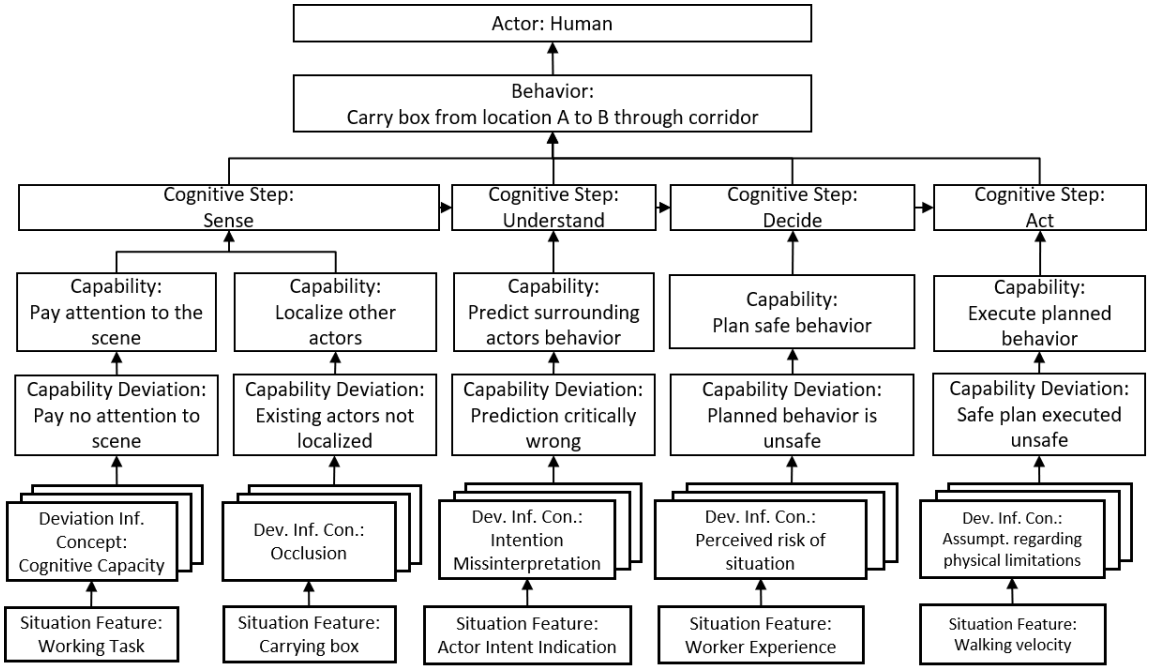


Figure 26 - The relation of behavior model elements from the ODE::SINADRA package (Figure 25) based on an example of a human tasked to carry a box from location A to B.

5.3 SAFETY OF MACHINE LEARNING

In this section, we present our proposals for extending the Event metamodel from Section 2.2 to address the issue of unexpected machine learning (ML) behaviour.

To establish the motivation for our proposal, we begin by providing an overview of related concepts of ML model evaluation with respect to situations where their outcomes diverge from expectation. As discussed further in section 5.3.1, there are several related concepts that should be concerned in terms of managing unexpected ML behaviour, and we propose our combined view over them, based on existing literature.

In Section 5.4, our concrete proposal for an extended set of event types to address the above aspects of ML uncertainty.

5.3.1 ML Uncertainty and related concepts

For ML models — and components incorporating them — to be effectively incorporated in dependability-critical systems, the fundamental challenge is to manage the effects of their outcomes when they diverge from expectation. As an example, how to manage cases where an ML model trained to identify a human face fails to do so during operation. Should the answer of the model be taken as is, and a critical decision depends on it, then a mistaken answer can lead to potentially unacceptable results.

While this view is not controversial in itself, the definition of uncertainty, and how it is quantified for ML models, offers several alternatives, and is closely related to the concepts of reliability and robustness. Due to the similarity between these concepts and their relevance for dependability, we find it is important to first clarify and establish our view on uncertainty, and how it relates to reliability and robustness, before discussing sources of each.

5.3.2 Definitions of Uncertainty

One relatively popular dimension of distinguishing uncertainty is the aleatoric/epistemic [9] [10] [11] [12] [13]. Across the literature, aleatoric tends to be interpreted as 'non-systematic' variability that is attributed to the inherent randomness of the underlying data across the space of operational situations. This does not consider the uncertainties arising from the model itself. Epistemic uncertainty is instead 'systematic', in the sense that it can be attributed to limitations in the models or the knowledge used to construct them. An example given in [10] establishes a coin flipping example, where the aleatoric uncertainty cannot be further reduced (inherent randomness of which side the coin lands on). Instead, epistemic uncertainty could be reduced if one were to consider additional information i.e. knowledge. Following the above example, if the coin in question was already weighed, measured, and already tested for its balance, one could reduce the corresponding epistemic uncertainty regarding a given coin toss.

As indicated earlier, numerous approaches already exist for quantifying uncertainty under the aleatoric/epistemic dimension, and this seems like a valid starting point. However, we find this distinction to be in some situations too coarse to be useful, particularly when managing dependability risk at runtime is concerned. This classification does not consider uncertainties arising due to the modelling process, and data management during the development of ML component. In a given runtime scenario, more details on the precise source of uncertainty could be leveraged so that the host application understands which course of action across several alternatives can be considered.

This leads us to the comparatively more refined view of the Uncertainty Wrapper (UW) model [14]. The UW decomposes an ML model's uncertainty in terms of a 3-layer 'onion' model, where:

- Model Fit is the innermost layer, capturing the inherent limitations of the learned ML model.
- Input Quality is the intermediate layer, capturing limitations introduced due to imperfection in the input data (e.g. faulty sensors).
- Scope Compliance is the outermost layer, addressing the difference between the trained scope and the target application scope of the given ML model.

The UW view allows us to not only distinguish between reducible and irreducible uncertainty (as per the aleatoric/epistemic dimension), but also enables more meaningful diagnosis and reactions to be planned out. For instance, aspects of model fitting can be evaluated during development of the model, whereas sources contributing to input quality and scope compliance can be accordingly isolated and managed independently. For instance, sources contributing to input quality loss may include sensor performance limitations or malfunction, and known methods such as sensor fusion [15] and measurement uncertainty models [16] can be considered there.

5.3.3 Definitions of Reliability

Reliability is a concept that is heavily used in the safety engineering domain and is incorporated into dependability as 'continuity of correct service' [17]. For an ML system, authors in [18] define reliability by any qualitative attribute related to a "vital

performance indicator" of that system. They included measures such as accuracy or inaccuracy, availability, downtime rate, responsiveness etc. In terms of ML, correctness can be established with respect to the ML model's answer being correct; for classification tasks, this means providing a true positive or true negative answer, and for regression tasks, this means providing an answer that is within some specific error bound.

The point of being able to relate ML reliability to this existing notion would allow us to incorporate such evaluation approaches within existing engineering approaches for reliable systems and services. Approaches for evaluating such reliability include [19] [20] [21] [22]. One point of note is that even across this limited selection of literature, the interpretation boundary between reliability and uncertainty is not always clear. Notably, in [22], the authors even incorporate robustness as part of their reliability evaluation.

5.3.4 Definitions of Robustness

Robustness is another popular concept in ML model evaluation literature, owing to the raised awareness of the related concept of adversarial attacks against ML models [23]. Robustness of ML models generally refers to evaluating the effect of relatively small deviations in model input on the model's output. Highly robust models are able to provide 'correct answers' (see section 5.3.3 on reliability) even under potentially significant deviation of the input.

A plethora of literature is available which can be generally summarized into two categories. One that focuses on improving robustness to dataset shift, and the second category that focuses on methods for evaluation of robustness itself. Robustness can be related to uncertainty by e.g. considering the robust optimization against distribution of model uncertainties [24]. Another solution is to consider a partially-characterized uncertain distribution, to obtain the robust optimization [25].

5.3.5 Unifying our views on uncertainty with existing engineering approaches

In [26], uncertainty is viewed in a more general sense as owing to the general notion of unexpected behaviour of software components. This view is particularly appropriate for encompassing ML uncertainty, as well as reliability and robustness, as it also accounts for the ecosystem of systems, components, services, as well as external interacting actors to the ML model itself. The authors distinguish between the following main sources of consideration regarding software uncertainty:

- Location, determining where in the overall architecture does the uncertainty manifest. This can be further decomposed into:
    - Model, encompassing aspects from section 5.3.2 for ML concerns.
    - Environment, due to imperfect expectation or awareness of the operational environment.
    - Goals, determining whether uncertainty is relating to existing application goals being or becoming invalid, or conflicting with external ones.
    - System complexity or state dynamicity.

- Resource dynamicity.

- Nature, with which the dimensions of uncertainty can be considered, e.g. aleatoric/epistemic or model fit/input quality/scope compliance.

- Determinism level, mainly distinguishing between qualitative and quantitative approaches for evaluation.

- Emergence timing, distinguishing between development and runtime emergence of uncertainty.

5.4 ODE EVENTS PROPOSAL

Figure 27 shows our initial proposal for introducing diagnosable/monitorable events for managing ML uncertainty. MLEvents are Events that may represent ML outcomes (correct/incorrect) and sources of uncertainty in terms of the UW view. ML event monitors can accordingly try to diagnose specific conditions of uncertainty, e.g. determining input quality or scope compliance. An UW monitor provides an ML model outcome with an uncertainty estimate, given a predefined confidence level. A scope compliance monitor, such as SafeML [27], can determine whether the model is still operating or has exited its trained scope, as indicated by its set of distance thresholds (optionally per ML feature dimension).

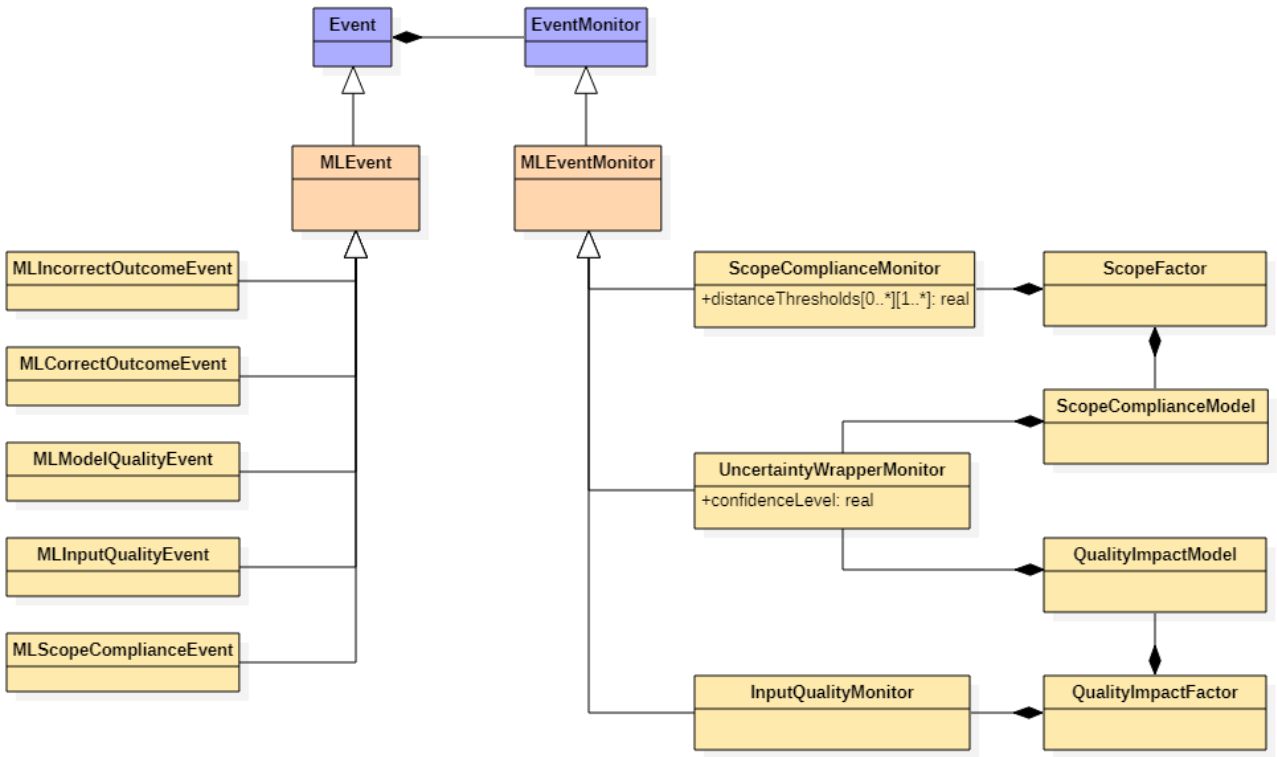


Figure 27 - ML Uncertainty metamodel

As the work on runtime assessment of uncertainty and ML safety/security is still in its early stages, the events above are only tentative at this point.

6. CONCLUSION

In this document, we have described the proposed new extensions to the ODE to support the EDDI concept and its associated executable activities at runtime. This involves changes to some existing packages and addition of some entirely new packages:

- A new ODE::Event package to define event-based communication with the host platform and other EDDIs at runtime. Events serve as feedback from the system via event monitors, while Actions act as recommendations from the EDDI back to the system in response to any detected dependability issues.

- Addition of an ODE::ConSert package (and sub-packages) to support the exchange of guarantees and demands over service provision in multi-agent or multi-robot systems.

- Changes to the ODE::FailureLogic package to further support dynamic dependability analysis, e.g. via the inclusion of non-failure causes and event sequences, as well as new links to security analysis.

- Changes to the ODE::FaultTree package to make fault trees more executable by including State information, Event triggers, and Action outcomes. Furthermore, because the FaultTree package is also used as the basis for security attack trees, both fault trees and attack trees are compatible and indeed interconnectable with each other. More on the co-engineering of safety and security can be found in **D4.3: Safety/Security Co-Engineering Framework**.

- Changes to the ODE::Markov package to make it into a more generic state machine package capable of modelling dynamic behaviour, with Events able to trigger state transitions and the possibility to fire Actions as a result.

- A new ODE::BayesianNetwork package to allow modelling of Bayesian networks. Like the previous analysis packages, this links to Events and Actions to enable input/output during runtime execution.

- Extensions to the TARA and FailureLogic packages to better support a more comprehensive security assessment framework, including new classes based on existing repositories of security vulnerabilities.

- Addition of a new ODE::SINADRA package to support situation-aware dynamic risk analysis at runtime.

- Specification of further events as preliminary support for assessment of ML reliability and robustness at runtime.

This updated ODE will allow the creation or modification of tools to generate EDDIs from design-time models that can then be executed at runtime.

7. REFERENCES

[1] R. Wei, T. Kelly, X. Dai, S. Zhao and R. Hawkins, "Model Based System Assurance Using the Structured Assurance Case Metamodel," *Journal of Systems and Software,* vol. 154, pp. 211-233, 2019.

[2] D. Schneider and M. Trapp, "Conditional Safety Certification of Open Adaptive Systems," *ACM Trans. Auton. Adapt. Syst. (ACM Transactions on Autonomous and Adaptive Systems),* vol. 8, no. 2, pp. 1-20, 2013.

[3] D. Schneider, "Conditional Safety Certification for Open Adaptive Systems," Technical University Kaiserslautern, 2014.

[4] W. E. Vesely, M. Stamatelatos, J. Dugan, J. Fragola, J. Minarick and J. Railsback, "Fault Tree Handbook with Aerospace Applications," NASA Office of Safety and Mission Assurance, USA, 2002.

[5] S. Kabir and Y. Papadopoulos, "Applications of Bayesian networks and Petri nets in safety, reliability, and risk assessments: a review," *Safety Science,* vol. 115, pp. 154-175, 2019.

[6] US Department of Defense, "MIL-P-1629: Procedure for Performing a Failure Mode, Effects and Criticality Analysis," US Department of Defense, Washington DC, USA, 1949.

[7] P. Feth, "Dynamic Behavior Risk Assessment for Autonomous Systems," Germany, 2020.

[8] J. Reich and M. Trapp, "SINADRA: Towards a Framework for Assurable Situation-Aware Dynamic Risk Assessment of Autonomous Vehicles," in *16th European Dependable Computing Conference (EDCC)*, Munich, Germany, 2020.

[9] D. Huseljic, B. Sick, M. Herde and D. Kottke, "Separation of aleatoric and epistemic uncertainty in deterministic deep neural networks," *2020 25th International Conference on Pattern Recognition (ICPR),* 2021.

[10] R. Senge, S. Bösner, K. Dembczyński, J. Haasenritter, O. Hirsch, N. Donner-Banzhoff and E. Hüllermeier, "Reliable classification: Learning classifiers that distinguish aleatoric and epistemic uncertainty," *Information Sciences,* vol. 255, 2014.

[11] J. Postels, H. Blum, Y. Strümpler, C. Cadena, R. Siegwart, L. Van Gool and F. Tombari, "The hidden uncertainty in a neural networks activations," *arXiv preprint arXiv:2012.03082,* 2020.

[12] E. Hüllermeier and W. Waegeman , "Aleatoric and epistemic uncertainty in machine learning: An introduction to concepts and methods," *Machine Learnin,* vol. 110, pp. 457-506.

[13] M. H. Shaker and E. Hüllermeier, "Aleatoric and Epistemic Uncertainty with Random Forests," *Advances in Intelligent Data Analysis XVIII,* pp. 444-456, 2020.

[14] M. Kläs and L. Sembach, "Uncertainty wrappers for data-driven models," *International Conference on Computer Safety, Reliability, and Security,* pp. 358-364, 2019.

[15] A. Assa and F. Janabi-Sharifi, "A Kalman filter-based framework for enhanced sensor fusion," *IEEE Sensors Journal,* vol. 15, pp. 3281-3292, 2015.

[16] B. Xiao, L. Cao, S. Xu and L. Liu, "Robust tracking control of robot manipulators with actuator faults and joint velocity measurement uncertainty," *IEEE/ASME Transactions on Mechatronics,* vol. 25, no. 3, pp. 1354-1365, 2020.

[17] A. Avizienis, J. C. Laprie, B. Randell and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing,* vol. 1, no. 1, pp. 11-33, 2004.

[18] Z. Bosnić and I. Kononenko, "An overview of advances in reliability estimation of individual predictions in machine learning," *Intelligent Data Analysis,* vol. 13, no. 2, pp. 385-401, 2009.

[19] S. Briesemeister, J. Rahnenführer and O. Kohlbacher, "No longer confidential: estimating the confidence of individual regression predictions," *PloS one,* vol. 7, no. 11, p. e48723, 2012.

[20] G. Adomavicius and Y. Wang, "Improving reliability estimation for individual numeric predictions: a machine learning approach," *INFORMS Journal on Computing,* 2021.

[21] P. Schulam and S. Saria, "Can you trust this prediction? Auditing pointwise reliability after learning," *The 22nd International Conference on Artificial Intelligence and Statistics,* pp. 1022-1031, 2019.

[22] X. Zhao, W. Huang, V. Bharti, Y. Dong, V. Cox, A. Banks, S. Wang, S. Schewe and X. Huang, "Reliability Assessment and Safety Arguments for Machine Learning Components in Assuring Learning-Enabled Autonomous Systems," *arXiv preprint arXiv:2112.00646,* 2021.

[23] S. Huang, N. Papernot, I. Goodfellow, Y. Duan and P. Abbeel, "Adversarial Attacks on Neural Network Policies," *arXiv preprint arXiv:1702.02284,* 2017.

[24] J. Goh and M. Sim, "Distributionally robust optimization and its tractable approximations," *Operations research,* vol. 58, no. 4-part-1, pp. 902-917, 2010.

[25] W. Chen and M. Sim, "Goal-driven optimization," *Operations Research*, vol. 57, no. 2, pp. 342-357, 2009.

[26] S. Mahdavi-Hezavehi, P. Avgeriou and D. Weyns, "A classification framework of uncertainty in architecture-based self-adaptive systems with multiple quality requirements," *Managing Trade-Offs in Adaptable Software Architectures*, pp. 45-77, 2017.

[27] K. Aslansefat, I. Sorokos, D. Whiting, R. Tavakoli Kolagari and Y. Papadopoulos, "SafeML: Safety monitoring of machine learning classifiers through statistical difference measures," *International Symposium on Model-Based Safety and Assessment*, pp. 197-211, 2020.

[28] US Department of Defense, "MIL-STD-1629A: Procedure for Performing a Failure Mode, Effects and Criticality Analysis," US Department of Defense, Washington DC, USA, 1980.

8. EXISTING ODE PACKAGES

8.1 OVERVIEW

Other than the packages covered previously, the ODE includes the following packages:

- ODE::Base
- ODE::Design
- ODE::HARA
- ODE::TARA
- ODE::Dependability (including Requirements and Domain sub-packages)
- ODE::Validation
- ODE::Integration

The ODE technically also includes the SACM packages, but those are not covered here.

The original ODE diagrams are reused but the colour scheme remains the same: yellow elements are part of the current package and blue are part of a different package.

8.2 ODE::BASE

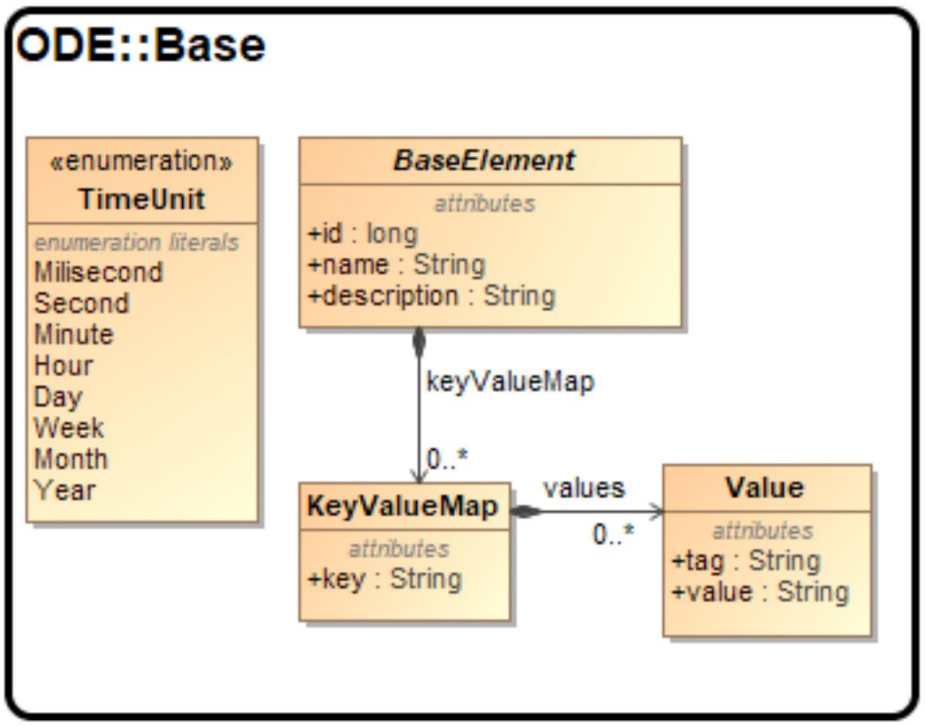


Figure 28 - ODE::Base package

The Base package covers fundamental entities like IDs and time units as well as an extensible key-value map structure that can be employed to create custom properties to store information not otherwise catered for by the rest of the metamodel. Most other modelling elements are subtypes of it.

ODE::Base::BaseElement

The root-most element, containing basic information common to all elements.

- *name:* The name of the element.
- *id:* A numeric ID number.
- *description:* Informal description of the element.
- *keyValueMap*[0..*]*:* An extensible KVM structure to hold arbitrary values.

ODE::Base::KeyValueMap

A typical key-value dictionary or map structure to hold user-defined properties.

- *key*: The key or index for a given value.
- *value[0..*]*: The value(s) corresponding to the key.

ODE::Base::Value

A user-defined value in the KVM.

- *tag:* A further description for specifying the value (e.g. in the case there are multiple values for a given key).
- *value:* The value itself.

8.3 ODE::DESIGN

The Design package is intended for system architecture modelling. It defines generic entities to describe systems, functions, components, ports to act as interfaces, and connections between them.

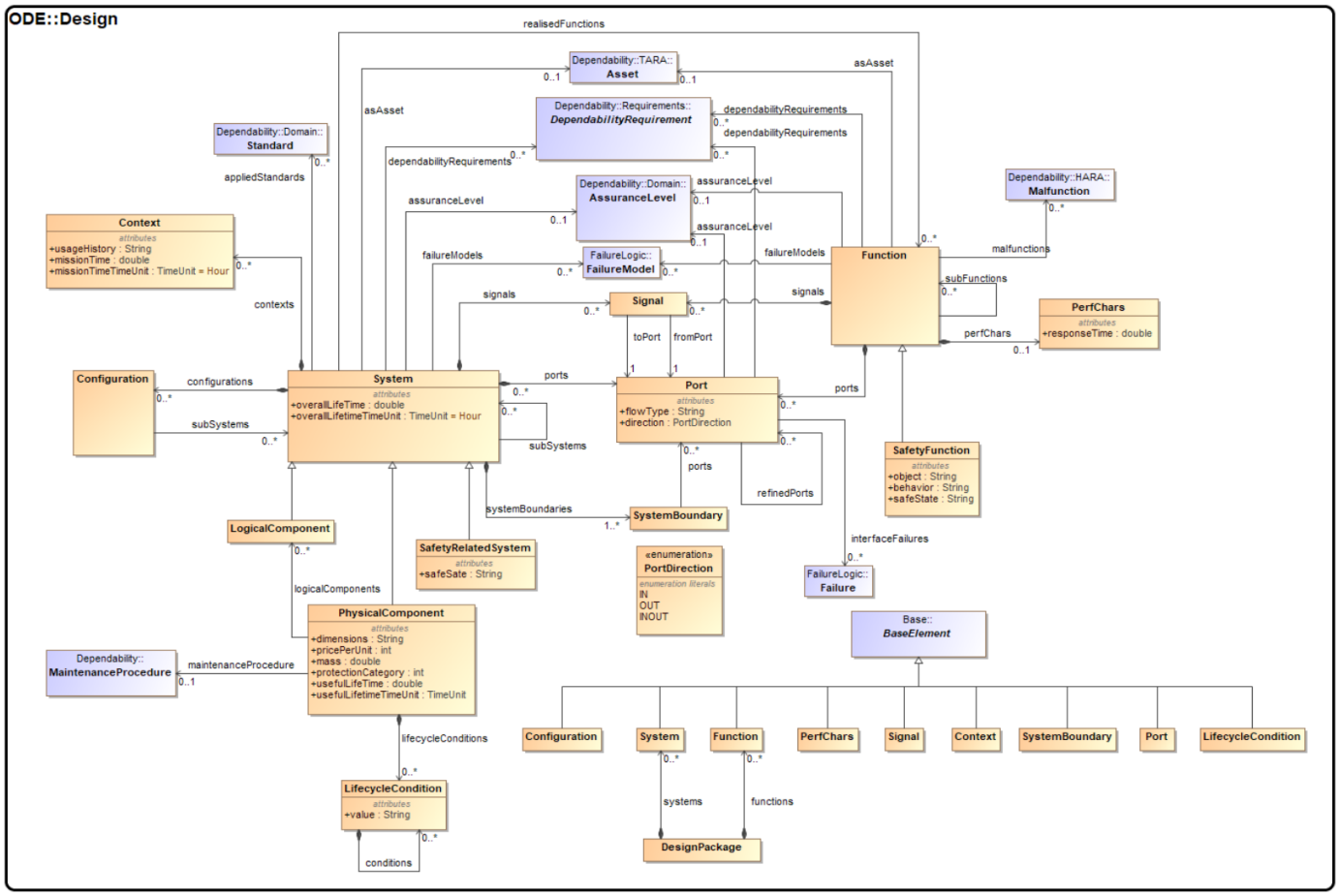


Figure 29 - ODE::Design package

Only the key elements are covered here.

ODE::Design::System

Represents a generic system and serves as the top-level entity for most of the other parts of the package. A System can have Ports and Signals (representing its external interface

and communication with the environment/other systems), as well as subsystems (to create a hierarchy). A System also maintains references to the failure models and possible dependability standards associated with the system as well as the functions it realises and requirements it fulfils.

- *overallLifeTime*: the expected amount of time the System will operate until being retired. Measured in overallLifetimeTimeUnits.
- *overallLifetimeTimeUnit*: the unit of time under which overall overallLifeTime is measured.
- *ports[0…*]:* The system's Ports. Can include physical and logical Ports.
- signals[0…*]: The Signals emanating from this system. For each such Signal, there's at least one Port owned by this system, with an 'out' direction, and set as the 'fromPort' of the Signal (see Constraints below).
- *subSystems[0…*]:* The Systems integrated under this System. This can include both physical and logical integration.
- *systemBoundaries[1…*]:* A grouping of Ports which represent a discrete interface of the System. At least one such grouping should exist, by default grouping all of the System's Ports under it.
- *configurations[0…*]:* The System's subsystems' Configurations.
- *contexts[0…*]:* The operational context(s) related to this System.
- *failureModels[0…*]:* The FailureModels associated with this System.
- *appliedStandards[0…*]:* The dependability Standards associated with this System.
- *realisedFunctions[0…*]:* The Functions this System is realising.
- *asAsset[0…1]:* A security-oriented interpretation of the System as an Asset to be protected.
- *dependabilityRequirements[0…*]:* DependabilityRequirements associated with this System.
- *assuranceLevel[0…1]:* An AssuranceLevel associated with this System's development.
- **[NEW]** *eventMonitors[0...*]:* The event monitors associated with this System.

ODE::Design::Context

Context describes the operational context of the System (particularly its expected mission time).

- *usageHistory*: log of system's previous usage.
- *missionTime*: system's expected/recorded operational period under this context.
- *missionTimeTimeUnit*: the time unit for missionTime.

ODE::Design::LogicalComponent

LogicalComponents can represent either conceptual or software components. Can only have other LogicalComponents as subsystems.

ODE::Design::PhysicalComponent

PhysicalComponents represent implementation in hardware. PhysicalComponents may also contain information on e.g. maintenance procedures and component lifetimes/lifecycle stages.

- *dimensions*: physical dimensions of the component (assumption: in S.I. units).
- *pricePerUnit*: economical cost of component.
- *mass*: mass of component (assumption: in S.I. units).
- *protectionCategory*: based on relevant protection standards, such as IP67.
- *usefulLifeTime*: the time period during which the component is expected/recorded to be functioning.
- *usefulLifeTimeTimeUnit*: the time unit for the usefulLifeTime.
- *lifecycleConditions*[0…*]: relevant conditions of lifecycle, such as storage condition, transportation condition etc.
- *maintenanceProcedure*[0…1]: the maintenance procedure planned/performed for this component.

ODE::Design::Port

Ports define the interface of a System or Component. They can have a variety of types (data, energy, fluid etc.) as well as a direction (input / output / both). Assurance levels (e.g. SILs) can be assigned to Ports as can dependability requirements and potential failures at that port.

- *flowType*: description of content (data, energy etc.) exchanged/expected through this port
- *direction*: direction of port's flow.
- *refinedPorts*[0…*]: constituent ports of a complex port.

- *interfaceFailures*[0…*]: system/function/component failures associated with the port.
- *assuranceLevel*[0…1]: Assurance Level assigned to Port during development.
- *dependabilityRequirements*[0…*]: DependabilityRequirements associated with Port.

ODE::Design::Function

A Function represents a conceptual or logical operation, thus supporting an earlier, more abstract functional architecture before a more detailed implementation becomes available. They too can contain sub-functions, Ports, requirements, failure models, and most of the other paraphernalia of a System.

- *subFunctions*[0…*]: constituent functions of a complex function.
- *ports*[0…*]: associated ports.
- *failureModels*[0…*]: failure analysis information associated with function.
- *malfunctions*[0…*]: malfunctions associated with a function.
- *perfChars*[0…1]: performance characteristics (PerfChars) of a function.
- *signals*[0…*]: signals leaving a Function.
- *assuranceLevel*[0…1]: Assurance Level assigned to Function during development.
- *dependabilityRequirements*[0…*]: Dependability Requirements associated with Function.

ODE::Design::Signal

Signals model links between two ports and typically represent some form of communication or transfer, though the actual role depends on the context. For example, a Signal between two Components could mean a service request / provision pair, but between a Component and a Function it could represent an implementation (i.e., the component implements the function).

- *fromPort*: origin port.
- *toPort*: target port.

8.4 ODE::DEPENDABILITY

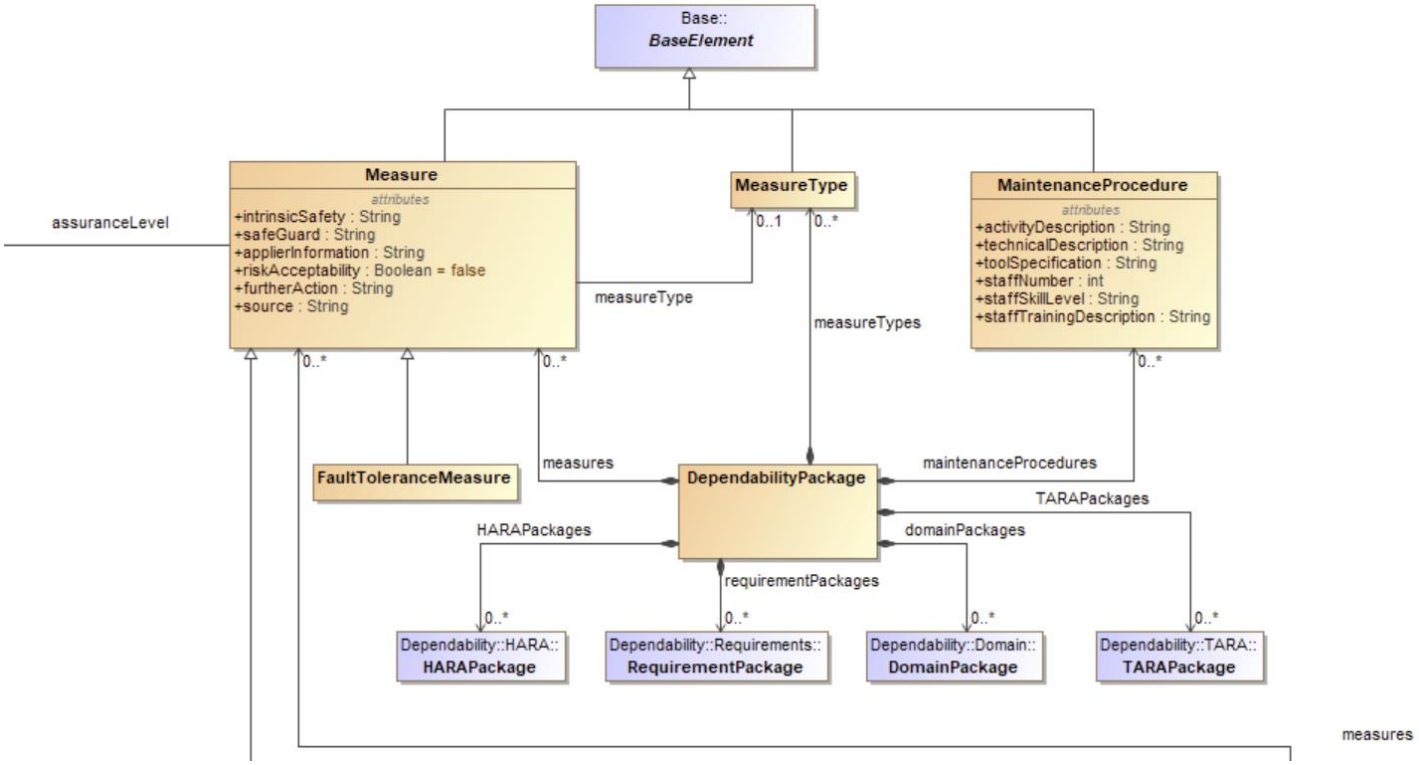


Figure 30 - ODE::Dependability package

The Dependability package is the overall package related to dependability requirements, standards, and risk analyses. The key element of the base Dependability package is the *Measure*, which represents a planned or applied measure to support dependability and/or ensure safety. The other major element is the *MaintenanceProcedure*.

ODE::Dependability::Measure

A planned or applied measure to support system dependability.

- *intrinsicSafety*: description of inherent safety provided by measure.
- *safeguard*: description of specific tolerance measure.
- *applierInformation*: safety manual for the user.
- *riskAcceptability*: whether the improved risk level is acceptable or not.
- *furtherAction*: whether any further measure after this failure tolerant measure is necessary or not.
- *source*: origin of the measure.
- *measureType*[0…1]: measure type description.
- *assuranceLevel*[0…1]: dependability assurance level of the measure.

ODE::Dependability::MaintenanceProcedure

A planned or applied maintenance procedure.

- *activityDescription*: basic description of what the procedure entails.
- *technicalDescription*: technical details regarding the procedure.
- *toolSpecification*: specification of the tool(s) used during the procedure.
- *staffNumber*: staff undertaking the procedure code/identifier.
- *staffSkillLevel*: discrete level or description of staff's technical qualification.
- *staffTrainingDescription*: detailed description of staff's training with regards to the procedure.

More extensive capabilities are provided by the various sub-packages: Domain, Requirements, HARA, and TARA.

8.5 ODE::DEPENDABILITY::DOMAIN

The Domain sub-package contains elements to describe relevant standards and definitions of assurance levels:

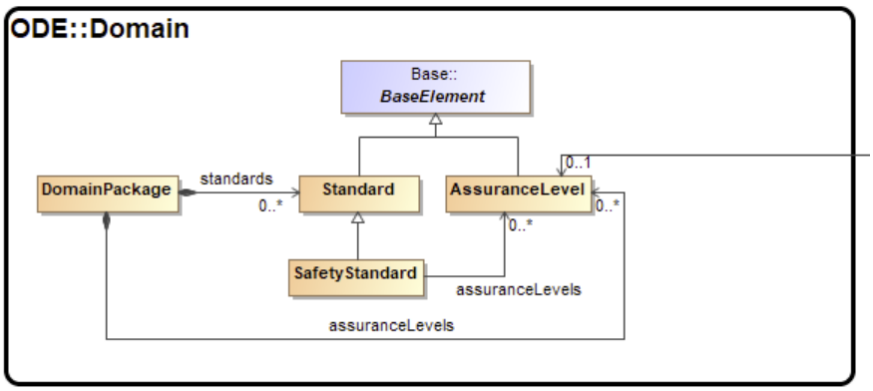


Figure 5 - ODE::Dependability::Domain

ODE::Dependability::Domain::Standard

Contains information about a relevant dependability standard.

ODE::Dependability::Domain::AssuranceLevel

A dependability assurance level, e.g. an ASIL, SIL, or DAL etc.

8.6 ODE::DEPENDABILITY::REQUIREMENTS

The second major sub-package is the Requirements package:

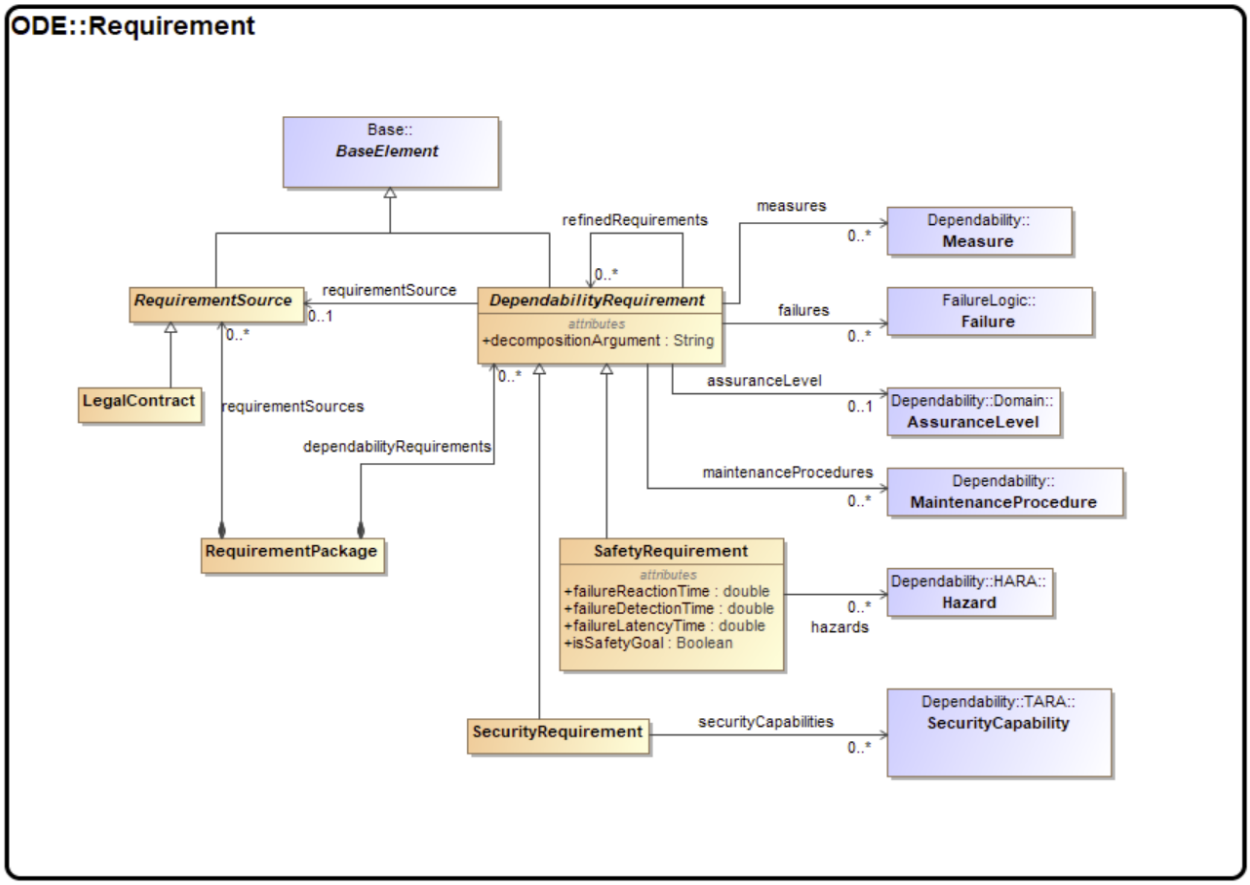


Figure 31 - ODE::Requirement package

The Requirement sub-package models dependability requirements (including both safety and security requirements). A Requirement can have a source and links to the failures and hazards that it is concerned with, the measures that address them, and any refined requirements that the Requirement can be decomposed into.

ODE::Dependability::RequirementPackage

Container for all Requirement elements.

- *dependabilityRequirements*[0…*]: contained DependabilityRequirements.
- *requirementSources*[0…*]: contained RequirementSources.

ODE::Dependability::RequirementSource

Represents the source from which a DependabilityRequirement element originates.

ODE::Dependability::LegalContract

Represents the source of DependabilityRequirements being a legal contract.

ODE::Dependability::DependabilityRequirement

A dependability requirement.

- *decompositionArgument*: describes how the DependabilityRequirement breaks down into further DependabilityRequirements.

- *requirementSource[0…1]*: describes the origin of the DepenabilityRequirement.

- *maintenanceProcedures[0…*]*: the MaintenanceProcedures planned/applied associated with this DependabilityRequirement.

- *measures[0…*]*: the planned/applied Measures associated with this DependabilityRequirement.

- *hazards[0…*]*: the Hazards addressed by this DependabilityRequirement.

- *failures[0…*]*: the Failure(Mode)s addressed by this DependabilityRequirement.

- *refinedRequirements[0…*]*: the DependabilityRequirements this DependabilityRequirement decomposes into.

- *requiredActions[0…*]*: the Actions associated with this requirement.

ODE::Dependability::SafetyRequirement

A safety requirement.

- *failureReactionTime*: the maximum reaction time after the detection of failure.

- *failureDetectionTime*: the maximum detection time after the occurrence of failure.

- *failureLatencyTime*: the maximum latency time until the failure/error is detected.

- *isSafetyGoal*: a safety goal is a top-level safety requirement. Here it is meant to distinguish whether the current requirement is a safety requirement or a top-level safety goal.

ODE::Dependability::SecurityRequirement

A security requirement.

- *capabilities[0…*]*: SecurityCapabilities associated with this SecurityRequirement.

8.7 ODE::DEPENDABILITY::HARA

The next sub-package is the HARA (Hazard Analysis and Risk Assessment) package:

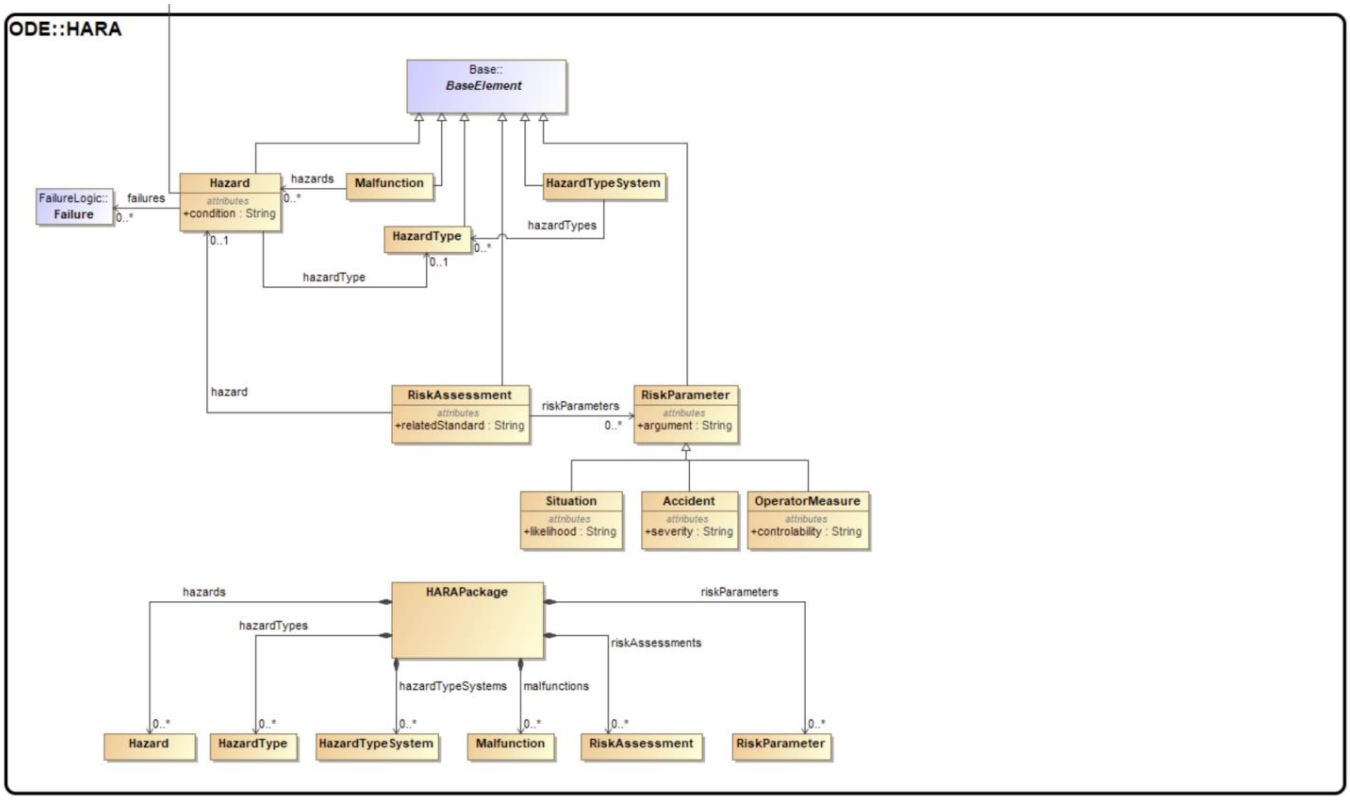


Figure 32 - ODE::HARA package

The HARA sub-package contains the necessary elements to support a hazard and risk analysis. The key elements here are the *Hazard*, which here specifically represents a situation involving loss of dependability with negative consequences, and the *Malfunction*, which is an unintended condition of a Function that causes one or more Hazards. Various risk parameters can be defined as part of the assessment and linked to the relevant hazards.

ODE::Dependability::HARA::Malfunction

An inadvertent state/behaviour of a Function likely leading to a Hazard.

- *hazards[0..*]:* Hazards caused by this malfunction.

ODE::Dependability::HARA::Hazard

A situation involving loss of safety leading to negative consequences.

- *condition*: Description of the Hazard's effects.
- *outputFailureMode[0..1]:* The System/Function output failure mode leading to the Hazard.
- *hazardType[0..1*]: The related hazard type.
- *measures[0..*]*: Dependability measures planned or applied to address or mitigate the Hazard.

ODE::Dependability::HARA::HazardType

Describes a Hazard's type. Can be complex and potentially derived from relevant standards or codified industrial practice.

ODE::Dependability::HARA::RiskAssessment

The result of the HARA. Identifies risk values based on Hazards and associated parameters.

- *hazard[0..1]:* The identified Hazard.
- *riskParameters[0..*]*: The collection of related risk parameters.

ODE::Dependability::HARA::Accident

The impact of a hazard.

- *severity:* A qualitative or quantitative measure of the impact.

ODE::Dependability::HARA::Situation

The likelihood of the hazard.

- *likelihood:* Qualitative or quantitative measure of probability of hazard occurrence.

8.8 ODE::DEPENDABILITY::TARA

Finally, the TARA sub-package is used for modelling a Threat Analysis and Risk Assessment. It includes key elements such as the Asset (an important concept or entity related to the operation of the system), Attacks (initiated by a ThreatAgent which threaten an Asset), and Vulnerabilities.

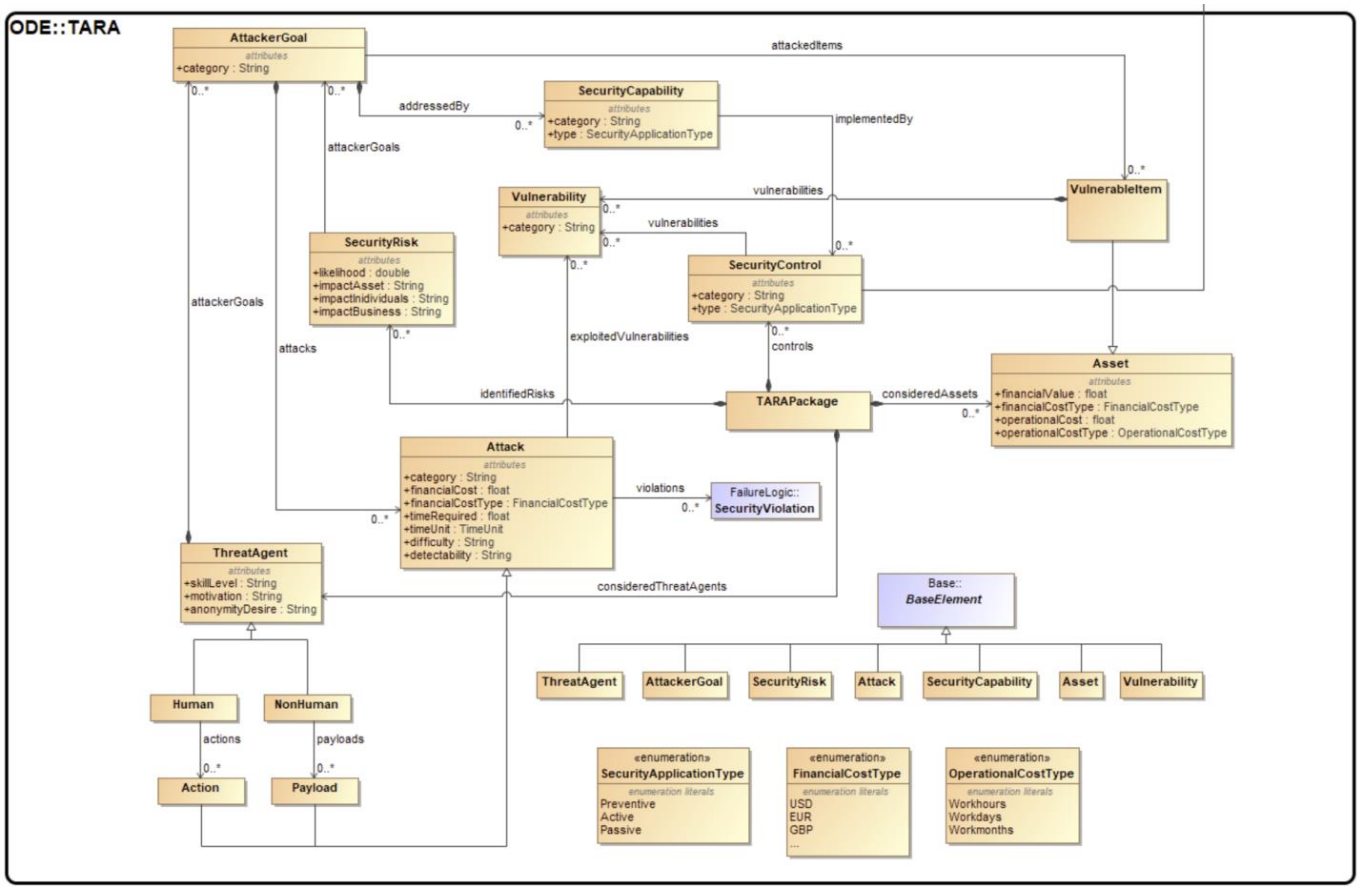


Figure 33 - ODE::TARA package

ODE::Dependability::TARA::TARAPackage

Entry point for collecting information encapsulating a (security) Threat Analysis and Risk Assessment (TARA).

- *consideredAssets[0…*]:* Assets that are subject to the given TARA.
- *identifiedRisks[0…*]:* Security Risks identified as part of this TARA.
- *consideredThreatAgents[0…*]*: Threat Agents considered as part of this TARA.
- *controls[0…*]:* Security Controls considered to address the Security Risks posed by the Threat Agents on the considered Assets.

ODE::Dependability::TARA::Asset

Represents a concept/entity related to the operation of the system that is valuable for its stakeholders. Often associated with financial values e.g. value lost during period of unavailability. Can include system elements.

- *financialValue*: Represents the financial cost of acquiring/owning the Asset; can be interpreted as the cost of loss, replacement, …

- *financialCostType*: The monetary unit financialValue is measured in e.g. US dollars, Euro, etc.

- *operationalCost*: Represents the cost in time of the Asset being unavailable wrt the system's operation; can be interpreted as the cost of operation halt, …

- *operationalCostType*: The time unit in lost workhours, days, or months.

ODE::Dependability::TARA:SecurityRisk

Represents a specific Security Risk, as assessed by the TARA captured in the associated TARAPackage. Risk combines a notion of likelihood of occurrence with a notion of severity of impact.

- *likelihood*: A numerical value representing the likelihood of occurrence (presumably during the ODE::Architecture::Context missionTime).

- *impactAsset*: Description of the worst-case impact of the Security Risk manifesting across all associated Assets.

- *impactIndividuals*: Description of the worst-case impact of the Security Risk manifesting across individuals e.g. pedestrians, users, general public…

- *impactBusiness*: Description of the worst-case impact of the Security Risk manifesting across the corporate system stakeholders e.g. developer company, customer company, …

- *attackerGoals[0…*]*: The (security) Attacker's Goals that, if successful, lead to the manifestation of the Security Risk.

ODE::Dependability::TARA:ThreatAgent

Represents an actor (physical or digital) whose Actions contribute to the potential manifestation of a Security Risk. Note that the actor may not always be malicious, sometimes just an unsuspecting participant e.g. via phishing attack.

- *skillLevel*: The necessary skill (knowledge, technical skill, …) of the security attacker such that his attack is successful.

- *motivation*: Description of the attacker's motivation, if any.

- *anonymityDesire*: Description of the attacker's desire to remain anonymous after executing his actions.

ODE::Dependability::TARA::Human

Represents a Human Threat Agent.

- *actions[0…*]:* The set of Actions (Attacks) the Agent can initiate.

ODE::Dependability::TARA::NonHuman
Represents a NonHuman i.e. digital Threat Agent.

- *payloads[0…*]*: The set of Payloads (Attacks) the Threat Agent can initiate.

ODE::Dependability::TARA::Attack

Represents an Attack initiated by a (security) Threat Agent.

- *category*: Description of the type of Attack e.g. 'Confidentiality', 'Integrity', …

- *financialCost*: The financial amount (estimate) required to launch the Attack e.g. cost of purchasing computational resources, …

- *financialCostType*: The monetary type in which the financial cost of launching the Attack is measured.

- *timeRequired*: The amount of time required to launch an Attack.

- *timeUnit*: The unit measuring the time required to launch an Attack.

- *difficulty*: Description of how difficult to execute successfully the Attack is.

- *detectability*: Description of how (easily) detectable the Attack is.

- *violations[0…*]*: The set of system-specific Security Violations potentially triggered by this Attack.

ODE::Dependability::TARA::AttackerGoal

Represents the objective of a ThreatAgent in attacking the system.

- *category*: Description of the kinds of security aspects this Goal involves e.g. 'Confidentiality', 'Integrity', …

- *attackedItems[0…*]*: Which system elements are subject to Attacks associated with this Goal.

- *attacks[0…*]*: Which Attacks are associated with this Goal.

- *addressedBy[0…*]*: Which Security Capabilities intend to address this Goal.

ODE::Dependability::TARA::SecurityCapability

Represents a high-level security requirement/objective.

- *category*: Description of the type of security issue this Capability addresses e.g. 'Confidentiality', 'Integrity', …

- *type*: Whether this Capability represents 'Preventive', 'Active', or 'Passive' measure(s).

- *implementedBy[0…*]*: The set of specific Security Controls implementing this Capability.

ODE::Dependability::TARA::SecurityControl

Represents a low-level security requirement. In contrast to the Security Capability representing comparatively higher-level security requirements. Also an ODE::Dependability::Measure

- *category*: Description of the type of security issue this Control addresses e.g. 'Confidentiality', 'Integrity', …
- *type*: Whether this Control represents 'Preventive', 'Active', or 'Passive' measure(s).
- *vulnerabilities[0…*]*: The set of Vulnerabilities addressed by this Security Control.

ODE::Dependability::TARA::Vulnerability

Represents a security weakness in the requirements, design, and/or implementation of the subject system.

- *category*: Description of the type of weakness this vulnerability involves e.g. 'Confidentiality', 'Integrity', …

ODE::Dependability::TARA::VulnerableItem

An Asset which features Vulnerabilities which potential Threat Agents could Attack.

- *vulnerabilities[0…*]*: The set of vulnerabilities this Asset is subject to.

8.9 ODE::VALIDATION

The Validation package specifies elements to model a testing-based approach to validation. This is built around the idea of TestExecutions that satisfy (or not) TestCases, which have a variety of TestParameters and AcceptanceCriteria associated with them.

8.10 ODE::INTEGRATION

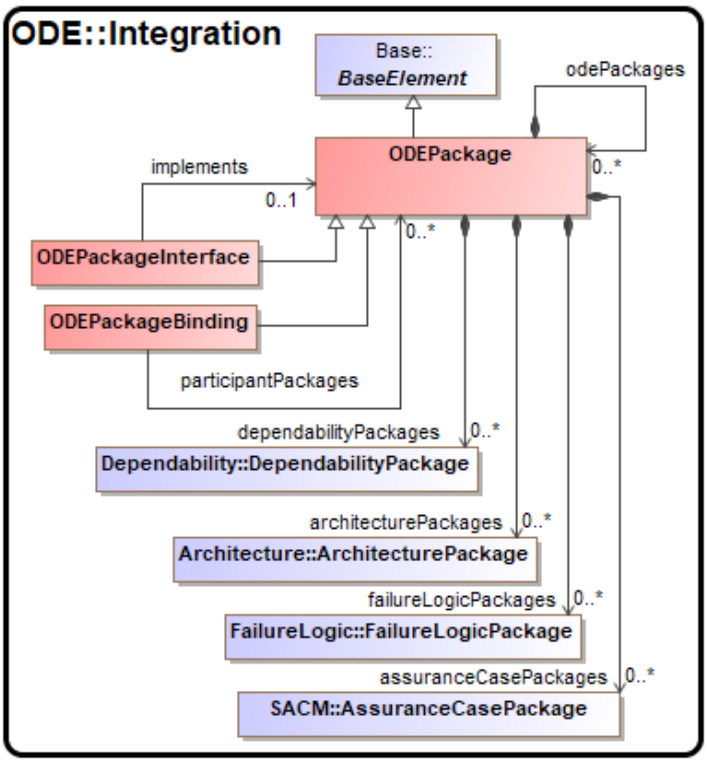


Figure 34 - ODE::Integration

Contains elements that allow other ODE packages (and potential extensions) to integrate cohesively as part of a single unit (i.e., a DDI or EDDI).