Application Protocol: Exchange of Thermal Model Data for Space Applications (STEP-TAS)

ESA reference	ESA-TEC-ST-002776
version	6.0
date	2018-02-09
download URI	http://ecss.nl/wp-content/uploads/2018/01/ecss-e-st-31-04-annex-c.zip

Table of contents

Foreword

This document describes the STEP-TAS data exchange protocol as a whole, and is derived from two previous documents which considered the protocol as being comprised of two parts:

- Part 1: Application protocol: Network-model and results format (STEP-NRF)
- Part 2: Application protocol: Thermal analysis for space (STEP-TAS)

The STEP-NRF part is an exchange protocol than can be used in its own right, and could potentially form a foundation layer for many other exchange protocols, not just STEP-TAS, and is the reason why two different documents were created. Other protocols would be able to reference STEP-NRF without needing STEP-TAS.

The original idea was to submit both documents describing the STEP-NRF and STEP-TAS exchange protocols for consideration by ISO (the International Organization for Standardization) for adoption as separate standards. STEP-NRF and STEP-TAS both follow the established ISO 10303 philosophy and use the same architectural building blocks, most notably for describing the protocol, the low-level data structures, and the exchange file representation. As as a consequence, the layout, format and terminology of the original documents describing the STEP-NRF and STEP-TAS protocols use the same conventions as the ISO 10303 standards.

The decision to submit STEP-TAS and STEP-NRF to ECSS instead of ISO provides the opportunity to review and harmonize these standards within one document. Some information about the ISO standardization process and standards will be retained in the document in order to provide context as to how and why the overall STEP-TAS exchange protocol is organized in the way it is.

Introduction

General

This protocol is based on ISO 10303. ISO 10303 is an International Standard for the computer-interpretable representation and exchange of product data. Its objective is to provide a neutral mechanism capable of describing product data throughout the life cycle of a product independent from any particular system. The nature of this description makes it suitable not only for neutral file exchange, but also as a basis for implementing and sharing product databases and long term archiving.

This protocol enables exchange and sharing of data needed for thermal control engineering of space products. It defines the context, scope, information requirements and a detailed formal computer-interpretable data model (in the form of an ISO 10303-11 EXPRESS schema) to support the exchange and sharing of all data needed in space thermal analysis and testing activities as well as the thermal aspects of the operation of space products.

The need for such a protocol has become evident in many space projects. Space industry is a domain in which very complex products are developed and operated by (usually large) international industrial teams. Analysis and test models are an essential part of the engineering process. It is not possible for the many partners in the industrial teams to standardise on the same tools for thermal analysis and test or operation results data processing. Nor is it desirable to do so, since healthy competition between the tool vendors promotes improvement and innovation at an affordable cost. At the same time an open standard that specifies an adequate neutral data format is the only viable way to realize reliable and cost effective data exchange and data sharing for the end-users: the thermal control engineers in the space industry.

Protocol and dictionary

This protocol is designed in such a way that all definitions in the data model (EXPRESS SCHEMA) are kept as generic as possible, so in the protocol itself no thermal control concepts are introduced. The full STEP-TAS data exchange standard consists of this protocol and a run-time loadable dictionary, that can be downloaded from a given URI. This dictionary contains a collection of pre-defined terms in the form of instances conforming to ENTITY specifications in this protocol's formal EXPRESS SCHEMA.

In order to use the standard an implementation first loads the STEP-TAS dictionary, which is a file in ISO 10303-21 format, and then uses the protocol plus the dictionary terms to create a STEP-TAS exchange dataset.

Splitting the standard into a core protocol and a run-time loadable dictionary has three major advantages:

1. The core protocol (EXPRESS SCHEMA) can be kept as simple as possible and free of particular space thermal engineering terminology. Its modules can be re-used without further adaptation by other space engineering disciplines such as space environment effects analysis.
2. Also the core protocol can remain unmodified for relatively long periods of time, because there is no need to produce new editions of the protocol for extension with simple new capabilities like adding support for a new physical property or a new type of node in a network model. This is good because the production and adoption of a new edition of an international standard is by its nature a time-consuming process, and usually necessitates upgrades of the software that implements the protocol, and in the worst case causes backwards incompatibility.

3. Last but not least, the run-time loadable dictionary can be extended as required through use of a light and flexible maintenance process. In almost all cases the extensions can be ensured to be backwards compatible with earlier releases of the dictionary and will not break existing software implementations.

1 Scope

1.1 Overall scope of STEP-TAS

In scope for this application protocol is all data that needs to be exchanged (or shared) between parties working in the domain of space thermal control engineering, over the whole life cycle of space products. The table below lists the main activities with typical input and output data. Typically these activities are run in an iterative incremental process, in which there is significant interaction with the other engineering and non-engineering disciplines needed to develop and operate a space product. Also there is substantial work in coordinating the analysis and test activities between different layers of the project team: the (international) supply chain consisting of a customer and a prime contractor with multiple tiers of subcontractors with and without engineering responsibility. The most important purpose of this protocol is indeed to help overcome the data exchange problems in the supply chain, by enabling smooth and reliable exchange and sharing of thermal control analysis and test models and results across the different levels of both project team and space system.

Table 1 – Main activities concerning space thermal control engineering

Activity	Input data	Output data
Specify thermal control design	system requirements, including operational space environment specification; properties of standard parts and materials; overall product structure; CAD specification; analysis results; test results; design data from prior projects; relevant standards, guidelines and handbooks;	thermal control part of product specification;
Define thermal control analysis and test cases	product specification; analysis models; test models;	analysis case specifications; test case specifications;
Construct thermal control analysis and test models	product specification; analysis case specifications; test case specifications;	analysis models; test models;
Do thermal radiative and conductive pre-processing	analysis models; analysis cases;	pre-processing results, such as radiative exchange factors, incident and absorbed environmental heat flows, thermal conductive network;
Do thermal network analysis	analysis models; analysis cases; thermal radiative and conductive pre-processing results;	analysis results, such as temperature fields and heat balances;
Run thermal tests	test article specification; test facility specification; sensor allocation specification; test procedures; test case specifications;	test results, such as time series of sensor readings;
Correlate analysis and test results	analysis results (predictions); test results (measurements); analysis model node to test sensor mapping specifications;	correlated analysis models; correlated analysis results;

The requirements for the data to be exchanged are defined in detail in:

— Clause 4 “Information Requirements”

1.2 Specific scope of STEP-NRF

The STEP-NRF protocol specifies the formal data structures necessary for the electronic exchange of network models and associated results data. A network model in this context is a generic representation of an engineering object (or a set of related engineering objects) by a collection of discrete network nodes and relationships between these nodes. The results are characteristic, predicted, assigned or observed quantities for components of the engineering object(s), which are sampled during an analysis, simulation, test or operation run.

This protocol is aimed at batch exchange of large amounts of results data, after completion of a run of the model. The protocol does not include specific provisions to handle streaming of results data during the execution of a run, although it is possible to address streaming in an implementation of this protocol.

The following are within the scope of this protocol:

— The representation of an engineering object by a network model of discrete nodes and relationships between the nodes, for the purpose of analysis, simulation, test or operation.
— Simple classification of network-models, discrete nodes and relationships between the nodes.
— The hierarchical breakdown of network-models, consisting of a root- model and a tree of submodels. There is no limit on the allowed number of breakdown levels.
— The definition and representation of scalar quantities and tensor quantities of any rank, including vector and matrix quantities. Scalar quantities may have the following value types: real, integer, character string or enumeration. All physical quantity values are associated with a unit.
— The definition and representation of properties of engineering objects.
— The definition and representation of analysis, simulation, test and operation runs with associated results.
— The definition and representation of a simple product structure, in the form of a product tree or an assembly tree, and the relationships between items defined in the product structure and items defined in the network-model representation.

The following are outside the scope of the STEP-NRF protocol:

— Discipline dependent specializations of network models, such as: thermal network models, structural finite element models, computational flow dynamics models.
— Geometric shape and topology definitions.

1.3 Fundamental concepts and assumptions of STEP-NRF

1.3.1 Activities relevant to analysis, simulation, test and operation

From an engineering perspective the following seven major categories of activity can be distinguished in the life cycle of a product:

1. *Definition* of the *mission* to be fulfilled, which is commonly called the *mission need statement*. This is then detailed into a full set of verifiable *requirements*, through a process of requirements

discovery and flowdown. The requirements specification starts as a systems engineering activity and is subsequently elaborated in full detail by each of the involved specialist disciplines.

2. *Definition* of the product *design* that fulfills the requirements specification. This is again a top-down activity coordinated by systems engineering and elaborated in each of the involved specialist disciplines. In general a phased approach is used with distinctive conceptual design, preliminary design and detailed design phases.
3. *Definition* of analysis, simulation, test or operation *cases*, and one or more *representative models* of the product of interest and its relevant environment. The *cases* define usage scenarios with relevant control parameters, initial and boundary conditions. The *cases* and *models* typically constitute idealizations of reality and combinations of expected extreme conditions in order to analyze, simulate or test a design rigourously against the specified requirements. Analysis or simulation is used to predict how the product will perform, in order to assist in design decisions and design option trade-offs. Analysis or simulation can also be used to verify that a given (detailed) product design meets its requirements in case physical test is impossible or too costly. Test is used to verify that the realized product (or an initial representative realization) operates conform the specified requirements. Observations from actual operation are used to measure how the product is performing while deployed in real use, and possibly to improve future designs for the same or similar kinds of product. An analysis, simulation, test or operation *case* may be broken down into a sequence of *subcases*. Similarly a *model* may need to be broken down into one or more *submodels*, in order to manage complexity, for instance by reflecting the hierarchical breakdown of a system in the form of a function tree, product tree or assembly tree.
4. *Execution* of analysis, simulation, test or operation *cases* and *models* , producing *results* data. One execution of a *case* and *model* is commonly called a *run*.
5. *Manufacturing*, *assembly* and *integration* of the product according to the design definitions.
6. *Delivery*, *deployment*, *operation* and *maintenance* of the product.
7. *Disposal* of the product.

In this protocol categories 3. and 4. are in scope, the other categories are not in scope.

The table below summarises the most important concepts that play a role in these two categories.

Table 2 – Most important data elements in the categories of activity relevant to analysis, simulation, test and operation

Category 3 – Definition of analyses, simulations, tests and operations	Category 4 – Execution of analyses, simulations, tests and operations
discrete network model (analysis, simulation, test, operation) case conceptual or detailed product structure quantity types and units material names and properties initial and boundary conditions parametric prescriptions user-defined logic	(analysis, simulation, test, operation) run results (produced per run) identification of the used tool or facility

1.3.2 The results datacube concept

The concept for storing results is very simple: results are stored as characteristic, predicted, assigned or observed quantities of observable items in a results space. In this results space, a property is identified by co-ordinates in only three dimensions:

— *what* is the *quantity type* for which a property is stored in the dataset?
— *where* in the model – for which *item* – does this property apply?
— *when* – that is: for which *state* – is a property observed?

The results space can be visualised as in Figure 1.

In the real world a number of factors complicate this simple concept:

— In order to interpret the results data correctly, an exchange dataset needs to provide an application and its user with sufficient (meta-)information *about* the exchange dataset and the properties.
— The results are not restricted to just bare numbers, so it is necessary to include information about *how* the results are represented. Even when the values of properties can be expressed as numeric scalars, it is necessary to include information about what these numbers signify. In the case of physical quantities one needs to know in which unit the values are expressed. Besides scalars, the values can also be vectors, tensors or descriptive data, for which representation structures have to be specified.

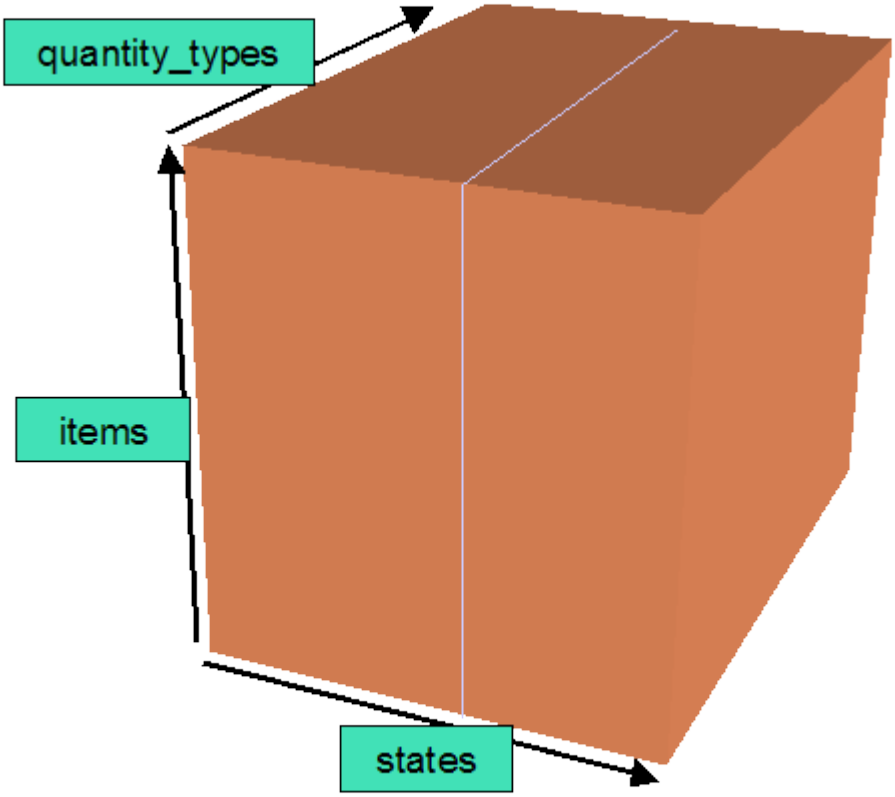

Figure 1 – Schematic illustration of the datacube and its three dimensions: observable items, quantity types and (discrete) states

A final consideration in the design of this protocol is that of efficiency, which is very important for bulk results data where models may contain many thousands of observable items, and the quantities for each item may be sampled thousands of times in a run, i.e. producing millions of results values. The straightforward approach of storing a potential property value for every property class, for every model component, at every possible state would lead to very inefficient storage because:

- Quantity types do not necessarily apply to all observable items.
- The observation of quantity values is not necessarily synchronised as the values are not always obtained for the same state (for example sample time). There is a good chance that it is impossible to determine the full state – which contains all applicable properties for all items – at a particular state of the represented engineering object.

Because of these reasons, valid and/or defined quantity values may be distributed very sparsely throughout the datacube.

The NRF protocol contains special features to achieve efficient storage, even for sparsely distributed data:

- the use of item lists prevent storage of data for meaningless combinations of items, quantity types and states,
- referencing common data structures (like text strings) prevents data duplication.

2 Normative references

IEC 60027-2	Letter symbols to be used in electrical technology - Part 2: Telecommunications and electronics
IETF/ RFC2396	"Uniform Resource Identifiers (URI): Generic Syntax", see http://www.ietf.org/rfc/rfc2396.txt
ISO 31	Quantities and units
ISO 10303-1	Industrial automation systems and integration — Product data representation and exchange — Part 1: Overview and fundamental principles
ISO 10303-11	Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual
ISO 10303-21	Industrial automation systems and integration — Product data representation and exchange — Part 21: Implementation methods: Clear text encoding of the exchange structure

3 Terms, definitions and abbreviations

For the purposes of this document, the terms and definitions given in ISO 10303 and the following apply.

3.1 Terms defined in ISO 10303-1

3.1.1
abstract test suite
a part of this International Standard that contains the set of abstract test cases necessary for conformance testing of an implementation of an application protocol.

3.1.2
application
a group of one or more processes creating or using product data.

3.1.3
application activity model (AAM)
a model that describes an application in terms of its processes and information flows.

3.1.4
application context
the environment in which the integrated resources are interpreted to support the use of product data in a specific application.

3.1.5
application interpreted model (AIM)
an information model that uses the integrated resources necessary to satisfy the information requirements and constraints of an application reference model, within an application protocol.

3.1.6
application object
an atomic element of an application reference model that defines a unique concept and contains attributes specifying the data elements of the object.

3.1.7
application protocol
a document complying with [AP-Guide] that specifies an application interpreted model satisfying the scope and information requirements for a specific application.

3.1.8
application reference model (ARM)
an information model that describes the information requirements and constraints of a specific application.

3.1.9
application resource
an integrated resource whose contents are related to a group of application contexts.

3.1.10
assembly

a product that is decomposable into a set of components or other assemblies from the perspective of a specific application.

3.1.11
component
a product that is not subject to decomposition from the perspective of a specific application.

3.1.12
conformance class
a subset of an application protocol for which conformance may be claimed.

3.1.13
conformance requirement
a precise, text definition of a characteristic required to be present in a conforming implementation.

3.1.14
data
a representation of information in a formal manner suitable for communication, interpretation, or processing by human beings or computers.

3.1.15
data exchange
the storing, accessing, transferring, and archiving of data.

3.1.16
data specification language
a set of rules for defining data and their relationships suitable for communication, interpretation, or processing by computers.

3.1.17
exchange structure
a computer-interpretable format used for storing, accessing, transferring, and archiving data.

3.1.18
generic resource
an integrated resource whose contents are context-independent.

3.1.19
implementation method
a part of this International Standard that specifies a technique used by computer systems to exchange product data that is described using the EXPRESS data specification language ISO 10303-11.

3.1.20
information
facts, concepts, or instructions.

3.1.21
information model
a formal model of a bounded set of facts, concepts or instructions to meet a specified requirement.

3.1.22
integrated resource

a part of this International Standard (that is ISO 10303) that defines a group of resource constructs used as the basis for product data.

3.1.23
interpretation
the process of adapting a resource construct from the integrated resources to satisfy a requirement of an application protocol. This may involve the addition of restrictions on attributes, the addition of constraints, the addition of relationships among resource constructs and application constructs, or all of the above.

3.1.24
PICS proforma
a standardised document in the form of a questionnaire, which, when completed for a particular implementation, becomes the protocol implementation conformance statement.

3.1.25
presentation
a recognisable visual representation of product data.

3.1.26
product
a thing or substance produced by a natural or artificial process.

3.1.27
product data
a representation of information about a product in a formal manner suitable for communication, interpretation, or processing by human beings or by computers.

3.1.28
product information
facts, concepts, or instructions about a product.

3.1.29
product information model
an information model which provides an abstract description of facts, concepts and instructions about a product.

3.1.30
protocol implementation conformance statement (PICS)
a statement of which capabilities and options are supported within an implementation of a given standard. This statement is produced by completing a PICS proforma.

3.1.31
resource construct
a collection of EXPRESS language entities, types, functions, rules and references that together define a valid description of an aspect of product data.

3.1.32
structure
a set of interrelated parts of any complex thing, and the relationships between them.

3.1.33
unit of functionality

a collection of application objects and their relationships that defines one of more concepts within the application context such that removal of any component would render the concept incomplete or ambiguous.

3.2 Terms Defined in ISO 10303-42

3.2.1
geometric coordinate system
the underlying global rectangular Cartesian coordinate system to which all geometry refers.

3.2.2
surface
a set of mathematical points which is the image of a continuous function defined over a connected subset of the plane (R^2). This image shall not be a single point or in part, or entirely, a curve.

3.3 Terms Defined in ISO 10303-45

3.3.1
material
the substance or substances of which product is composed or made.

3.3.2
material designation
an identifier which is assigned by agreement.

3.3.3
material property
a product characteristic which depends upon the material or materials comprising the product.

3.4 Terms defined in the STEP-NRF protocol

3.4.1
case
named collection of purpose, parameters, initial conditions, boundary conditions and sequencing information that specifies how a model shall be executed

NOTE In this protocol "case" is used as a generalized term encompassing the specializations: "analysis case", "simulation case", "test case" and "product operation case".

NOTE The definition of a case may also contain pass and/or fail criteria for successful execution.

3.4.2
class
named category of items which share common characteristics and behaviour

3.4.3
datacube
collection of discrete quantity values, structured in a three-dimensional mathematical space with discrete lists of observable items, quantity types and states as bases

3.4.4
derived quantity category
named subcategory of quantities of the same kind that is a specialisation of a principal quantity category

3.4.5
exchange dataset
coherent and valid set of instances of entities conforming to the schema defined in this protocol

NOTE The dataset is instantiated in the form of a physical file, such as ISO 10303-21, a working form (e.g. in computer memory) or as part of a (shareable) database. An exchange dataset can be accessed through a programming interface, such as the SDAI defined in ISO 10303-22.

3.4.6
identifier
character string that uniquely identifies an item in a context

NOTE The context in which the item is thus uniquely identified is often referred to as the *identifier scope* or *namespace*. The identifier does not need to have a natural language meaning but be a computer sensible encoding. The term identifier is often referred to by the abbreviation *id*. In many applications the terms *number* or *key* are used as synonyms for identifier.

3.4.7
model
representation in software or hardware of one or more aspects of a product, and possibly its operational environment, for the purpose of design, analysis, simulation or verification

NOTE A model is typically an idealised representation of a product.

3.4.8
name
character string with natural-language meaning by which something can be referred to

NOTE A name is a string that represents the human-interpretable name of something. Often a name does not uniquely identify an item. In many applications the terms "label" or "tag" are used as synonyms for name.

3.4.9
observable item
item of which one or more properties can be observed

3.4.10
principal quantity category
named top level category of quantities of the same kind

NOTE Principal for a quantity category means that it constitutes a base or top level category from which other quantity categories may be derived.

EXAMPLE An example of a principal quantity category is "length" which is also a base quantity in ISO 31. To "length" a number of derived quantity categories may be associated such as "distance", "diameter", "radius" and "width".

3.4.11
property
quality or attribute belonging and especially peculiar to an item, common to all members of a class

NOTE 1 Definition derived from [Webster].

NOTE 2 In this protocol "property" is specifically used in the meaning of a "quantity" for an "observable item" in a certain "state".

3.4.12
qualifier
word or word group that limits or modifies the meaning of another word or word group [Webster]

NOTE In this protocol "qualifier" is specifically used to qualify the meaning of a "quantity".

3.4.13
quantity
number or character-string valued characteristic of something

NOTE In this protocol the definition of "quantity" is extended beyond the more restrictive meaning of physical quantity (i.e. a number valued characteristic) as defined in ISO 31.

3.4.14
root-model
highest level model in a hierarchical model breakdown

3.4.15
run
named execution of an analysis, simulation, test or operation model and case

NOTE A "run" has a clearly defined start and end. A "run" is the execution of a model and a case using some tool or facility. For analysis or simulation this is typically a software tool, for test this is a test facility and for operation this is the operational environment itself. A "run" produces analysis, simulation, test or operation results. Such a collection of results is identified by the identifier, name, timestamp and similar attributes of a run.

3.4.16
sensor
means or device for observation of a quantity

3.4.17
state
mode or condition of being [Webster]

NOTE In this protocol a "state" is identified by a state quantity value. The state quantity type is typically (discrete sampled) time or frequency, but can also be a string valued quantity type.

EXAMPLE An example of a string valued quantity type is "product life-time category" with values "Beginning of Life" and "End of Life".

3.4.18
submodel
next lower level model in a hierarchical model breakdown

NOTE A submodel may contain other (yet lower level) submodels. The model breakdown is a whole-part composite tree structure that forms an acyclic graph.

3.4.19
supermodel
next higher level model in a hierarchical model breakdown

NOTE A supermodel contains one or more submodels. The model breakdown is a whole-part composite tree structure that forms an acyclic graph.

3.4.20
tensor
generalization of the concept of vector that consists of a set of components usually having multiple rows of indices that are functions of the co-ordinate system and have invariant properties under transformation of the co-ordinate system [Webster]

3.4.21
Unicode base character
character that does not graphically combine with preceding characters, and that is neither a control nor a format character.

NOTE Definition D13 from Unicode standard v4.0, Section 3.6.

3.4.22
unit
reference quantity with numerical value one

NOTE Definition from ISO 31-0.

3.5 Terms defined in the STEP-TAS protocol

3.5.1
apoapsis
orbit position farthest away from the celestial body that governs the orbit

3.5.2
multiple reflection
thermal radiation propagating from one thermal-radiative face to another thermal-radiative face via at least one intermediate reflection

3.5.3
orbit
trajectory in space, governed by gravitational forces and/or propulsion

3.5.4
orbit arc
part of an orbit.

3.5.5
periapsis
orbit position nearest to the celestial body that governs the orbit

3.5.6
radiative coupling

thermal mathematical network model conductor that specifies thermal-radiative heat transfer between two thermal-radiative faces i and j

NOTE The radiative coupling between two thermal-radiative faces i and j is equal to the product of the surface area of face i and the radiative exchange factor from i to j and the emissivity of face i, or – alternatively – the product of the surface area of face j and the radiative exchange factor from j to i and the emissivity of face j. The physical dimension of a radiative coupling is L^2.

3.5.7
radiative exchange factor
fraction of the thermal radiation between two thermal-radiative faces i and j that is emitted by face i and that is finally – possibly after multiple reflections – absorbed by face j

3.5.8
rigid body
rigid kinematic object

NOTE A rigid body is a collection of parts of a product which can not move one relative to another. The collection of parts is considered to be rigid at least for the purpose of engineering analysis. As a consequence movement is only possible at interfaces between rigid bodies. The boundaries of a rigid body coincide with the boundaries of a (compound) meshed geometric item.

3.5.9
sensor
means or device for observation of a quantity during test or operation

3.5.10
space mission
activity performed or to be performed by a space product

NOTE Aspects of a space mission comprise the orbit trajectories, attitudes, space environment, operational modes and sequence of events. A space mission may be subdivided into mission phases.

3.5.11
thermal network model
thermal mathematical model defined by a network of discrete nodes, where the physical properties of the network nodes are considered to be uniform

NOTE A thermal network node is considered to be iso-thermal and has uniform thermo-optical and thermo-physical properties. The model comprises a nodal breakdown and a network of links between the nodes. Such a model is usually called a thermal network model, because of the analogy with an electrical circuit network schema. The purpose of the model is to represent a product for the purpose of thermal analysis computations.

3.5.12
thermal node
network node, that is the discretization atom in a thermal network model

3.5.13
thermal-radiative face
part of one side of a bounded surface, that is used as a discretization atom in a thermal- radiative model.

NOTE A thermal-radiative face has thermo-optical properties and represents a the part of a surface where thermal-radiative heat transfer can occur. In many space thermal analysis tools thermal-radiative faces are also used to define thermal- conductive (pseudo-)solids by assigning a notional thickness to the face as well as thermo-physical properties.

3.5.14
thermal-radiative model
geometrical surface model representing a product for the purpose of thermal-radiation analysis, in particular for computation of radiative exchange factors and environmental thermal heat flows

3.5.15
viewfactor
fraction that specifies the measure for the direct view from one face to another face

NOTE The viewfactor from face i to face j is defined as the quotient of the solid angle representing the direct view from i to j over the solid angle of total possible view from face i. For a flat face the total possible view is a solid angle of 2π steradians, i.e. a hemisphere. The viewfactor is also often referred to as the *geometric viewfactor*.

3.6 Abbreviations

AAM	Application Activity Model
AIM	Application Interpreted Model
AP	Application Protocol
ARM	Application Reference Model
B-rep	Boundary Representation Solid Model
CAD	Computer Aided Design
CNES	Centre National d'Etudes Spatiales – French Space Agency
ESA	European Space Agency
ESTEC	ESA's European Space Research and Technology Centre
IETF	Internet Engineering Task Force (http://www.ietf.org)
ISO	International Organization for Standardization
MGM	Meshed Geometric Model
NRF	Network-model and Results Format
OO	Object-Oriented
RFC	Request For Comments (IETF)
SDAI	Standard Data Access Interface (defined in [STEP-22])
SET	*Standard d'Echange et de Transfert* (French Standard AFNOR Z68-300)
SET-ATS	SET Protocol d'Application Thermique Spatiale
SI	*Système International des Unités* - International System of Units
SKM	Space Kinematic Model
SMA	Space Mission Aspects
STEP	Standard for the Exchange of Product Model Data (ISO 10303)
TAS	Thermal Analysis for Space
TBD	To be defined
UoF	Unit of Functionality
URD	User Requirements Document
URI	Uniform Resource Identifier (specified in IETF/RFC2396, see http://www.ietf.org/rfc/rfc2396.txt)

4 Information requirements

This clause specifies in detail what kind of data needs to be exchanged in the domain of discrete engineering models and associated results. The engineering models are restricted to those that can be represented by a network of discrete nodes and relationships between the nodes. The information requirements are grouped in three subclauses: Units of Functionality, Application Objects and Application Assertions.

4.1 Modular breakdown of the protocol

The protocol is composed of the following four modules:

1. Network-model and results format (NRF) module, defined below in clause 4.2;
2. Meshed geometric model (MGM) module, defined below in clause 4.3;
3. Space kinematic model (SKM) module, defined below in clause 4.4;
4. Space mission aspects (SMA) module, defined below in clause 4.5.

Each module extends the previous one. The complete STEP-TAS protocol contains all four modules and is defined in clause 4.5.

4.2 Network-model and results format (NRF) module

This subclause specifies the units of functionality for the Network-model and Results Format Application Protocols. This protocol specifies the following units of functionality:

1. NRF General support
2. NRF Quantities and units
3. NRF Date and time
4. NRF Parametrics
5. NRF Network model representation
6. NRF Cases, runs and results
7. NRF Product structure
8. Materials

The elements that each UoF supports are given below. The application objects included in the UoFs are defined in subclause 4.2.

4.2.1 SCHEMA declaration for nrf_arm

The following EXPRESS declaration begins the nrf_arm schema.

Express specification:

```
SCHEMA nrf_arm;
-- $Id$
```



```
-- Copyright (c) 1995-2018 European Space Agency (ESA).
-- All rights reserved.
```

4.2.2 CONSTANT declaration

The following EXPRESS specification declares the global constants.

Express specification:

```
CONSTANT
  SCHEMA_OBJECT_IDENTIFIER : STRING :=
    '{http://www.purl.org/ESA/step-tas/v6.0/nrf_arm.exp}';
  -- in formal ISO version to be replaced with
  -- '{ iso standard n part(p) version(v) }'
END_CONSTANT;
```

Constant definitions:

— SCHEMA_OBJECT_IDENTIFIER provides a built-in way to reference the object identifier of the protocol for version verification. For the definition and usage of the object identifier see ISO 10303-1 and Annex E.

4.2.3 NRF General support UoF

The general_support UoF specifies application objects that are used to support the use of the principal application objects that are listed in the other Units of Functionality of this subclause.

NOTE The objects in this UoF are taken as much as possible literally from ISO 10303-41 and ISO 10303-203. An "nrf_" tag is prefixed to all CONSTANT, TYPE, ENTITY, RULE, FUNCTION and PROCEDURE identifiers.

4.2.3.1 TYPE nrf_identifier

An **nrf_identifier** is a string suitable for identifying some data.

NOTE Typically the *id* attribute of type **nrf_identifier** is used to uniquely identify an entity instance within a given context.

Express specification:

```
TYPE nrf_identifier = STRING;
WHERE
  is_valid_identifier: nrf_verify_identifier(SELF);
END_TYPE;
```

Formal propositions:

— is_valid_identifier: the identifier string may only contain alpha-numeric, underscore, hyphen or dot characters and may not start with a hyphen or dot.

4.2.3.2 FUNCTION nrf_verify_identifier

The function **nrf_verify_identifier** verifies the value of an nrf_identifier. It returns TRUE if the provided identifier string only contains alpha- numeric, underscore, hyphen, or dot characters and does not start with a hyphen or dot. Otherwise FALSE is returned.

Express specification:

```
FUNCTION nrf_verify_identifier(a_string : STRING) : BOOLEAN;
  IF LENGTH(a_string) < 1 THEN
    RETURN(FALSE);
  END_IF;
  IF (a_string[1] = '-') OR (a_string[1] = '.') THEN
    RETURN(FALSE);
  END_IF;
  REPEAT i := 1 TO LENGTH(a_string);
    IF NOT(
      {'A' <= a_string[i] <= 'Z'} OR
      {'a' <= a_string[i] <= 'z'} OR
      {'0' <= a_string[i] <= '9'} OR
      (a_string[i] = '_') OR
      (a_string[i] = '-') OR
      (a_string[i] = '.')) THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— a_string specifies the candidate identifier to be verified.

4.2.3.3 TYPE nrf_uniform_resource_identifier

An **nrf_uniform_resource_identifier** is a URI string suitable to identify some product data resource.

Express specification:

```
TYPE nrf_uniform_resource_identifier = STRING;
END_TYPE;
```

Informal propositions:

— ip1: The string value shall conform to IETF/RFC2396.

4.2.3.4 TYPE nrf_label

An **nrf_label** is the term by which something can be referred to. It is a string that represents the human-interpretable name of something and shall have a natural-language meaning. An **nrf_label** may contain any Unicode base characters.

Express specification:

```
TYPE nrf_label = STRING;
WHERE
```



```
  is_valid_label: nrf_verify_label(SELF);
END_TYPE;
```

Formal propositions:

— is_valid_label: verify that the label does not contain control characters as defined in ISO-IEC 10646 / Unicode.

4.2.3.5 TYPE nrf_non_blank_label

An **nrf_non_blank_label** is an nrf_label that contains at least one character and does not start with a space.

Express specification:

```
TYPE nrf_non_blank_label = nrf_label;
WHERE
  is_valid_non_blank_label: (LENGTH(SELF) > 0) AND (SELF[1] <> ' ');
END_TYPE;
```

Formal propositions:

— is_valid_non_blank_label: The label contains at least one character and does not start with a space.

4.2.3.6 FUNCTION nrf_verify_label

The function **nrf_verify_label** verifies the value of an nrf_label. The function returns TRUE if the provided nrf_label contains only non-control characters as defined in ISO-IEC 10646 / Unicode. Otherwise FALSE is returned.

Express specification:

```
FUNCTION nrf_verify_label(a_string : STRING) : BOOLEAN;
  REPEAT i := 1 TO LENGTH(a_string);
    IF NOT(
      {' ' <= a_string[i] <= '~'} OR
      (a_string[i] >= "000000A0")) THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— a_string specifies the candidate label to be verified.

4.2.3.7 TYPE nrf_text

An **nrf_text** is a string of characters that is intended to be read and understood by a human being. It has information purposes only.

Express specification:


```
TYPE nrf_text = STRING;
END_TYPE;
```

4.2.3.8 TYPE nrf_positive_integer

An **nrf_positive_integer** is an INTEGER value greater than zero.

Express specification:

```
TYPE nrf_positive_integer = INTEGER;
WHERE
  is_positive: SELF > 0;
END_TYPE;
```

Formal propositions:

- is_positive: the value shall be greater than zero

4.2.3.9 TYPE nrf_non_negative_integer

An **nrf_non_negative_integer** is an INTEGER value greater than or equal to zero.

Express specification:

```
TYPE nrf_non_negative_integer = INTEGER;
WHERE
  is_non_negative: SELF >= 0;
END_TYPE;
```

Formal propositions:

- is_non_negative: the value shall be greater than or equal to zero.

4.2.3.10 TYPE nrf_negative_integer

An **nrf_negative_integer** is an INTEGER value less than zero.

Express specification:

```
TYPE nrf_negative_integer = INTEGER;
WHERE
  is_negative: SELF < 0;
END_TYPE;
```

Formal propositions:

- is_negative: the value shall be less than zero.

4.2.3.11 ENTITY nrf_address

An **nrf_address** is the information necessary for communicating, using one or more communication methods.

Express specification:

```
ENTITY nrf_address
  SUPERTYPE OF(ONEOF(nrf_organizational_address, nrf_personal_address));
  internal_location       : OPTIONAL nrf_label;
  street_number           : OPTIONAL nrf_label;
  street                  : OPTIONAL nrf_label;
  postal_box              : OPTIONAL nrf_label;
  town                    : OPTIONAL nrf_label;
  region                  : OPTIONAL nrf_label;
  postal_code             : OPTIONAL nrf_label;
  country                 : OPTIONAL nrf_label;
  facsimile_number        : OPTIONAL nrf_label;
  telephone_number        : OPTIONAL nrf_label;
  electronic_mail_address : OPTIONAL nrf_label;
  telex_number            : OPTIONAL nrf_label;
WHERE
  has_at_least_one_attribute:
    EXISTS(internal_location) OR
    EXISTS(street_number) OR
    EXISTS(street) OR
    EXISTS(postal_box) OR
    EXISTS(town) OR
    EXISTS(region) OR
    EXISTS(postal_code) OR
    EXISTS(country) OR
    EXISTS(facsimile_number) OR
    EXISTS(telephone_number) OR
    EXISTS(electronic_mail_address) OR
    EXISTS(telex_number);
END_ENTITY;
```

Attribute definitions:

- internal_location optionally specifies an organization-defined address for internal mail delivery.
- street_number optionally specifies the number of a location on a street.
- street optionally specifies the name of a street.
- postal_box optionally specifies the number of a postal box.
- town optionally specifies the name of a town.
- region optionally specifies the name of a region.
- postal_code optionally specifies the code that is used by the country's postal service.
- country optionally specifies the name of a country.
- facsimile_number optionally specifies the number at which facsimiles may be received.
- telephone_number optionally specifies the number at which telephone calls may be received.
- electronic_mail_address optionally specifies the electronic address at which electronic mail may be received.
- telex_number optionally specifies the number at which telex messages may be received.

Formal propositions:

- has_at_least_one_attribute: At least one of the attributes shall have a value.

4.2.3.12 ENTITY nrf_organization

An **nrf_organization** is an administrative structure.

NOTE This definition is re-used from ISO 10303-41.

Express specification:

```
ENTITY nrf_organization;
  id          : nrf_identifier;
  name        : nrf_label;
  description : nrf_text;
UNIQUE
  has_unique_name: name;
END_ENTITY;
```

Attribute definitions:

- id specifies the identifier of an instance of **nrf_organization**.
- name specifies the human-interpretable name of an instance of **nrf_organization**, that is the name by which the organization is known.
- description specifies the textual description of an instance of **nrf_organization**.

Formal propositions:

- has_unique_name: The name of an **nrf_organization** shall be unique in a dataset.

4.2.3.13 ENTITY nrf_organizational_address

An **nrf_organizational_address** is an address for one or more organizations.

NOTE This definition is re-used from ISO 10303-41.

Express specification:

```
ENTITY nrf_organizational_address
  SUBTYPE OF(nrf_address);
  organizations : SET [1:?] OF nrf_organization;
  description   : nrf_text;
END_ENTITY;
```

Attribute definitions:

- organizations specifies the organizations located at the address.
- description specifies the textual description of an instance of **nrf_organizational_address**.

4.2.3.14 ENTITY nrf_organizational_project

An **nrf_organizational_project** is a project for which one or more organizations are responsible.

Express specification:


```
ENTITY nrf_organizational_project;
  name                       : nrf_label;
  description                : nrf_text;
  responsible_organizations : SET [1:?] OF nrf_organization;
END_ENTITY;
```

Attribute definitions:

- name specifies the human-interpretable name of an instance of **nrf_organizational_project**.
- description specifies the textual description of an instance of **nrf_organizational_project**.
- responsible_organizations specifies the nrf_organization(s) who are responsible for the project.

4.2.3.15 ENTITY nrf_person

An **nrf_person** represents an individual human being.

Express specification:

```
ENTITY nrf_person;
  id            : nrf_identifier;
  last_name     : OPTIONAL nrf_label;
  first_name    : OPTIONAL nrf_label;
  middle_names  : OPTIONAL LIST [1:?] OF nrf_label;
  prefix_titles : OPTIONAL LIST [1:?] OF nrf_label;
  suffix_titles : OPTIONAL LIST [1:?] OF nrf_label;
UNIQUE
  has_unique_id: id;
WHERE
  has_first_or_last_name: EXISTS(last_name) OR EXISTS(first_name);
END_ENTITY;
```

Attribute definitions:

- id specifies the identifier of an instance of **nrf_person**, which distinguishes the person.
- last_name optionally specifies a person's last name.
- first_name optionally specifies a person's first name.
- middle_names optionally specifies a person's middle name(s).
- prefix_titles optionally specifies the person's social or professional standing and appear before his or her name(s).
- suffix_titles optionally specifies the person's social or professional standing and appear after his or her name(s).

Formal propositions:

- has_unique_id: The **nrf_person**'s id shall be unique in the dataset.
- has_first_or_last_name: Either the last name, the first name or both shall be defined.

4.2.3.16 ENTITY nrf_person_and_organization

An **nrf_person_and_organization** represents a person in an organization.

Express specification:


```
ENTITY nrf_person_and_organization;
  the_person       : nrf_person;
  the_organization : nrf_organization;
END_ENTITY;
```

Attribute definitions:

- the_person specifies a person who is related to an organization.
- the_organization specifies the organization to which the person is related.

4.2.3.17 ENTITY nrf_personal_address

An **nrf_personal_address** is an address for one or more persons.

Express specification:

```
ENTITY nrf_personal_address
  SUBTYPE OF(nrf_address);
  people      : SET [1:?] OF nrf_person;
  description : nrf_text;
END_ENTITY;
```

Attribute definitions:

- people specifies the people who reside at the address.
- description specifies the textual description of an instance of **nrf_personal_address**.

4.2.3.18 ENTITY nrf_approval

An **nrf_approval** is a confirmation of the quality of the product data that it is related to.

Express specification:

```
ENTITY nrf_approval;
  level                  : nrf_label;
  date_time              : nrf_date_and_time;
  by_person_organization : nrf_person_and_organization;
END_ENTITY;
```

Attribute definitions:

- level specifies a type or level of approval in terms of its usage.
- date_time specifies the date and time at which the approval was given.
- by_person_organization specifies the person and organization by whom the approval was given.

4.2.3.19 ENTITY nrf_tool_or_facility

An **nrf_tool_or_facility** specifies the identification of a (software) tool or a facility (e.g. a test facility) that was used to produce (part of) the data in an exchange dataset.

Express specification:

```
ENTITY nrf_tool_or_facility;
  name        : nrf_label;
  description : nrf_text;
  owner       : OPTIONAL nrf_organization;
UNIQUE
  has_unique_name: name;
END_ENTITY;
```

Attribute definitions:

- name specifies the name of the tool or facility, when applicable including its version.
- description specifies the textual description of the tool or facility.
- owner optionally specifies the organization that owns or manages the tool or facility.

Formal propositions:

- has_unique_name: the name shall be unique in the dataset.

4.2.3.20 ENTITY nrf_security_classification_level

An **nrf_security_classification_level** is a specification of a category of security for access to data.

NOTE **nrf_security_classification_level** is adapted from ISO 10303-41.

EXAMPLE The following strings are typical security classification level identifiers:
'unclassified', 'classified', 'proprietary', 'confidential', 'secret', 'top_secret'

Express specification:

```
ENTITY nrf_security_classification_level;
  id          : nrf_identifier;
  name        : nrf_label;
  description : nrf_text;
UNIQUE
  has_unique_id: id;
END_ENTITY;
```

Attribute definitions:

- id specifies the identifier of an **nrf_security_classification_level**.
- name specifies the human-interpretable name of an **nrf_security_classification_level**.
- description specifies the textual description of an **nrf_security_classification_level**.

Formal propositions:

- has_unique_id: the id shall be unique in the dataset.

4.2.4 NRF Quantities and units UoF

The quantities_and_units UoF specifies application objects that capture the definition of physical and non-physical quantities, SI and non-SI units, and aggregated quantities in the form of vectors, matrices and tensors.

The guiding reference for these specifications is ISO 31-0, which says in its first Clause: "This part of ISO 31 gives general information about principles concerning physical quantities, equations, quantity and unit symbols, and coherent unit systems, especially the International System of Units, SI."

ISO 31-0 Clause 2.1 "Physical quantity, unit and numerical value" continues with:

> In ISO 31 only physical quantities used for the quantitative description of physical phenomena are treated. Conventional scales, such as the Beaufort scale, Richter scale and colour intensity scales, and quantities expressed as the results of conventional tests, e.g. corrosion resistance, are not treated here, neither are currencies nor information contents.
>
> Physical quantities may be grouped together into categories of quantities which are mutually comparable. Lengths, diameters, distances, heights, wavelengths and so on would constitute such a category. Mutually comparable quantities are called "quantities of the same kind".
>
> If a particular example of a quantity from such a category is chosen as a reference quantity called the *unit*, then any other quantity from this category can be expressed in terms of this unit, as a product of this unit and a number. This number is called the *numerical value* of the quantity expressed in this unit.
>
> In formal treatment of quantities and units, this relation may be expressed in the form: $\mathbf{A} = \{\mathbf{A}\} \cdot [\mathbf{A}]$ where $\mathbf{A}$ is the symbol for the physical quantity, $[\mathbf{A}]$ the symbol for the unit and $\{\mathbf{A}\}$ symbolizes the numerical value of the quantity $\mathbf{A}$ expressed in the unit $[\mathbf{A}]$. For vectors and tensors the components are quantities which may be expressed as described above.

In this protocol the concept of quantities and units is generalized as described hereafter.

A distinction is made between *scalar* and *tensor* quantities. All quantities must have a unique name and a (preferrably) unique symbol. A *scalar quantity* has a single value, possibly with uncertainty margin(s). A *tensor quantity* is a compound quantity containing a number of scalar quantity elements that are accessed through indices along one or more dimensions.

A *physical quantity* is a scalar quantity with a *numerical value* that denotes *magnitude* or *multitude*. It must also have an associated *unit*. The *unit* defines the scale on which to interpret the *numerical value*. The *datatype* of the value of a *physical quantity* is a real number when expressing *magnitude*, and an integer number when expressing *multitude*.

The *physical dimensions* of a *physical quantity* are specified through seven *dimensional exponents*, one for each of the seven SI base quantities: *length*, *mass*, *time*, *electric current*, *thermodynamic temperature*, *amount of substance* and *luminous intensity*. A quantity may be *dimensionless*, in which case all its dimensional exponents are zero.

A *physical quantity category* is a named category of physical quantities of the same kind, i.e. mutually comparable quantities in ISO 31-0 terminology. All quantities that belong to one *physical quantity category* must have – by definition – the same *physical dimensions*.

A *quantity category qualifier* is a qualifier that is used to further specialize the meaning and usage of a *physical quantity category*. One or more qualifiers may be applied to create such a specialized *physical quantity category*.

A *unit* is a named reference quantity for a particular physical quantity category, with numerical value one, that is standardized by some standardization body. A *unit* has a (unique) symbol in addition to its name. A *dimensionless quantity* always has the special unit *one*, denoting the mathematical number 1, as defined in ISO 31. SI units may be defined using *multiple* or *submultiple* prefixes such as *milli*, *kilo*, *mega*, etc. In this

protocol the SI units are extended with the binary data units *byte* and *bit* as defined in IEC 60027-2 and the binary data *multiple* prefixes *kibi*, *mebi*, *gibi* , *tebi*, etc.

Besides the physical quantity, also two kinds of *non-physical quantity* are introduced: *enumeration quantity* and *string quantity*. An *enumeration quantity* has a value that is selected from a given (finite) enumeration of character strings and a *string quantity* may have any character string value.

A *quantity type* defines the kind of quantity and the datatype in which the quantity values will be stored. A *physical quantity type* must also specify the applicable *unit* and it may optionally specify an *uncertainty margin specification method*. A (scalar) *quantity* can therefore be considered to consist of the combination of a *value* – possibly with uncertainty margins – and a *quantity type*. A *physical quantity type* is always associated with one *physical quantity category*.

As already mentioned a *tensor quantity* is a multi-dimensional, compound quantity built up from a number of scalar quantity elements. The number of dimensions is called the *rank* of the tensor. A rank one tensor is a *vector* and a rank two tensor is a *matrix*. Rank three and higher tensors are called *higher order tensors*. The number of elements in a *tensor quantity* is equal to the product of its dimensions. Each of the elements of a tensor quantity may be given a separate name. In this protocol the concept of tensor is taken beyond its strict mathematical definition in the sense that its elements may be non-physical quantities: enumeration or string quantities as described earlier.

NOTE 1 The nrf_quantities_and_units definitions take into account the measure_schema of ISO 10303-41, but simplify and generalize those definitions at the same time. In particular the entity inheritance tree has been rationalized to need single inheritance only and avoid any complex entity instances as in the original ISO 10303-41 definitions. Care has been taken to ensure that a one-to- one mapping of the entities defined here to those of ISO 10303-41 remains straightforward.

NOTE 2 Conventional scales, such as the Beaufort scale, Richter scale and colour intensity scales, and quantities expressed as the results of conventional tests, e.g. corrosion resistance (as quoted above from ISO 31-0), can be captured in the form of nrf_context_dependent_unit instances.

4.2.4.1 TYPE nrf_unit_symbol_identifier

An **nrf_unit_symbol_identifier** is a string suitable for identifying the symbol for a unit.

Express specification:

```
TYPE nrf_unit_symbol_identifier = STRING;
WHERE
  has_valid_characters: nrf_verify_unit_symbol_identifier(SELF);
END_TYPE;
```

Formal propositions:

— has_valid_characters: only alpha-numeric, dot, circonflex, minus sign, slash, left or right parenthesis characters are allowed within an **nrf_unit_symbol_identifier**.

4.2.4.2 FUNCTION nrf_verify_unit_symbol_identifier

The function **nrf_verify_unit_symbol_identifier** verifies the value of an nrf_unit_symbol_identifier. It returns TRUE if the unit symbol string only contains alpha-numeric, dot, circonflex, minus sign, slash, left or right parenthesis characters. Otherwise FALSE is returned.

Express specification:

```
FUNCTION nrf_verify_unit_symbol_identifier(a_symbol : STRING) : BOOLEAN;
  IF LENGTH(a_symbol) < 1 THEN
    RETURN(FALSE);
  END_IF;
  REPEAT i := 1 TO LENGTH(a_symbol);
    IF NOT(
      {'A' <= a_symbol[i] <= 'Z'} OR
      {'a' <= a_symbol[i] <= 'z'} OR
      {'0' <= a_symbol[i] <= '9'} OR
      (a_symbol[i] = '.') OR
      (a_symbol[i] = '^') OR
      (a_symbol[i] = '-') OR
      (a_symbol[i] = '/') OR
      (a_symbol[i] = '(') OR
      (a_symbol[i] = ')')) THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— a_symbol specifies the candidate unit symbol identifier to be verified.

4.2.4.3 ENTITY nrf_any_unit

The **nrf_any_unit** is an abstract supertype that provides a generic mechanism to reference any unit. Each **nrf_any_unit** is either an nrf_base_unit, an nrf_context_dependent_unit or an nrf_derived_unit.

Express specification:

```
ENTITY nrf_any_unit
  ABSTRACT SUPERTYPE OF(ONEOF(
    nrf_base_unit,
    nrf_context_dependent_unit,
    nrf_derived_unit));
  name                : STRING;
  symbol              : nrf_unit_symbol_identifier;
  quantity_category   : nrf_primary_physical_quantity_category;
UNIQUE
  has_unique_name: name;
  has_unique_symbol: symbol;
END_ENTITY;
```

Attribute definitions:

— name specifies the name of a unit.

— symbol specifies the short symbol string through which a unit may be presented when a short notation is appropriate.

— quantity_category specifies the nrf_primary_physical_quantity_category to which the unit is related. Through the quantity_category the dimensional exponents for the unit are defined in terms of the seven SI base quantities as defined in ISO 31.

Formal propositions:

— has_unique_name: the name shall be unique in the dataset.

— has_unique_symbol: the symbol shall be unique in the dataset.

4.2.4.4 ENTITY nrf_base_unit

An **nrf_base_unit** is a type of nrf_any_unit that provides a generic mechanism to reference an nrf_extended_si_unit or an nrf_conversion_based_unit. It is a single base unit with exponent one.

Express specification:

```
ENTITY nrf_base_unit
  ABSTRACT SUPERTYPE OF(ONEOF(
    nrf_extended_si_unit,
    nrf_conversion_based_unit))
  SUBTYPE OF(nrf_any_unit);
END_ENTITY;
```

4.2.4.5 ENTITY nrf_extended_si_unit

An **nrf_extended_si_unit** is a type of nrf_base_unit that specifies an SI unit complying with one of the units defined in ISO 31 or a binary data unit complying with one of the units defined in IEC 60027-2.

NOTE The prefixes for binary data units are listed in the table below.

Factor	Name	Symbol	Origin	Corresponding SI prefix
2^{10}	kibi	Ki	kilobinary: $(2^{10})^1$	kilo: $(10^3)^1$
2^{20}	mebi	Mi	megabinary: $(2^{10})^2$	mega: $(10^3)^2$
2^{30}	gibi	Gi	gigabinary: $(2^{10})^3$	giga: $(10^3)^3$
2^{40}	tebi	Ti	terabinary: $(2^{10})^4$	tera: $(10^3)^4$
2^{50}	pebi	Pi	petabinary: $(2^{10})^5$	peta: $(10^3)^5$
2^{60}	exbi	Ei	exabinary: $(2^{10})^6$	exa: $(10^3)^6$

Express specification:

```
ENTITY nrf_extended_si_unit
  SUBTYPE OF(nrf_base_unit);
  prefix    : STRING;
  base_name : STRING;
DERIVE
  SELF\nrf_any_unit.name : nrf_non_blank_label := prefix + base_name;
  SELF\nrf_any_unit.symbol : nrf_unit_symbol_identifier :=
    nrf_derive_extended_si_symbol(SELF);
  prefix_factor : REAL := nrf_derive_extended_si_prefix_factor(prefix);
WHERE
  has_valid_prefix: prefix IN [
    '','yotta','zetta','exa','peta','tera','giga','mega','kilo','hecto','deca',
    'deci','centi','milli','micro','nano','pico','femto','atto','zepto','yocto',
    'exbi','pebi','tebi','gibi','mebi','kibi'];
  has_valid_name_and_dimensional_exponents:
    nrf_verify_base_name_and_exponents_for_extended_si_unit(SELF);
END_ENTITY;
```

Attribute definitions:

- prefix specifies the multiple or submultiple name as defined in ISO 31 and IEC 60027-2. This defines the multiplication factor for the unit. It may be the empty string. An empty string implies a multiplication factor of one.
- base_name specifies the SI or IEC 60027-2 base name of an **nrf_extended_si_unit**.
- name is derived from the concatenation of prefix and base_name.
- symbol is derived from the concatenation of the prefix symbol and base_name symbol as defined in ISO 31 and IEC 60027-2.
- prefix_factor is the multiplication factor that is derived from the prefix string.

Formal propositions:

- has_valid_prefix: the prefix shall be specified as the empty string or it shall be one of the prefixes for multiples or submultiples as defined in ISO 31 and IEC 60027-2.
- has_valid_name_and_dimensional_exponents: the base_name shall be one of the SI unit names defined in ISO 31 or 'bit' or 'byte' as defined in IEC 60027-2, and shall have the matching dimensional exponents.

4.2.4.6 ENTITY nrf_conversion_based_unit

An **nrf_conversion_based_unit** is a type of nrf_base_unit that specifies a unit that is defined through a linear conversion relationship with respect to a reference unit. The unit conversion relationship is defined by the following equation:

$y_{converted} = factor \cdot x_{reference} + offset$

where:

$y_{converted}$ is the quantity expressed in the **nrf_conversion_based_unit**, and,

$x_{reference}$ is the same quantity expressed in the reference unit.

The reference unit may be another **nrf_conversion_based_unit** or an nrf_extended_si_unit. Ultimately every **nrf_conversion_based_unit** is defined with respect to an nrf_extended_si_unit.

EXAMPLE 1 An **nrf_conversion_based_unit** for inch is specified with the name 'inch', where the reference_unit is an nrf_extended_si_unit with empty prefix '' and name 'metre', the factor = 1.0/0.0254 and the offset = 0.0.

EXAMPLE 2 An **nrf_conversion_based_unit** for degree Fahrenheit is specified with the name = 'degree_Fahrenheit', where the reference_unit is an nrf_extended_si_unit for 'kelvin', the factor = 1.8 and the offset = -459.67.

Express specification:

```
ENTITY nrf_conversion_based_unit
  SUBTYPE OF(nrf_base_unit);
  reference_unit : nrf_base_unit;
  factor         : REAL;
  offset         : REAL;
DERIVE
  SELF\nrf_any_unit.quantity_category : nrf_primary_physical_quantity_category :=
    reference_unit.quantity_category;
WHERE
  has_no_circular_reference_unit_dependency:
    nrf_verify_no_circular_reference_unit_dependency(SELF);
END_ENTITY;
```

Attribute definitions:

- reference_unit specifies the unit to be used as the reference for the definition of the conversion based unit.
- factor specifies the multiplication factor for the unit conversion relationship.
- offset specifies the offset for the unit conversion relationship.
- quantity_category is redeclared and derived to be the same as the quantity_category of the reference_unit.

Formal propositions:

- has_no_circular_reference_unit_dependency: The chain of reference_unit dependencies shall not be circular.

4.2.4.7 FUNCTION nrf_verify_no_circular_reference_unit_dependency

The function **nrf_verify_no_circular_reference_unit_dependency** verifies that the chain of reference unit dependencies is not circular. The function returns TRUE when no circular dependency exists and FALSE otherwise.

Express specification:

```
FUNCTION nrf_verify_no_circular_reference_unit_dependency(
    a_conversion_based_unit : nrf_conversion_based_unit) : BOOLEAN;
  LOCAL
    the_reference_unit : nrf_base_unit;
  END_LOCAL;
  the_reference_unit := a_conversion_based_unit.reference_unit;
  REPEAT WHILE 'NRF_ARM.NRF_CONVERSION_BASED_UNIT' IN TYPEOF(the_reference_unit);
    IF the_reference_unit :=: a_conversion_based_unit THEN
      RETURN(FALSE);
    END_IF;
    the_reference_unit := the_reference_unit.reference_unit;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- a_conversion_based_unit specifies an nrf_conversion_based_unit to be verified.

4.2.4.8 ENTITY nrf_context_dependent_unit

An **nrf_context_dependent_unit** is a type of nrf_any_unit which is not related to the system of units defined in this standard, i.e. such a unit is not ultimately related to a combination of the seven SI base units.

EXAMPLE 1 The number of parts in an assembly is a physical quantity that may be measured in a unit called "component". Such a unit cannot be related to an SI unit.

EXAMPLE 2 Also conventional scales, such as the Beaufort scale, Richter scale and colour intensity scales, and units for quantities expressed as the results of conventional tests (as quoted above from ISO 31-0), can be captured in the form of **nrf_context_dependent_unit** instances, e.g. with name "Beaufort_Scale", "Richter_Scale", etc.

Express specification:


```
ENTITY nrf_context_dependent_unit
  SUBTYPE OF(nrf_any_unit);
END_ENTITY;
```

4.2.4.9 ENTITY nrf_derived_unit

An **nrf_derived_unit** is a type of nrf_any_unit which specifies a unit defined by an expression of other base units, possibly raised to an exponent.

EXAMPLE The unit "newton per square millimetre" is a derived unit which would be defined using an nrf_derived_unit_element referencing an nrf_extended_si_unit for "newton" with an exponent equal to 1.0 and an nrf_derived_unit_element referencing an nrf_extended_si_unit for "milli" "metre" with an exponent equal to –2.0.

Express specification:

```
ENTITY nrf_derived_unit
  SUBTYPE OF(nrf_any_unit);
  elements                : LIST [1:?] OF nrf_derived_unit_element;
DERIVE
  SELF\nrf_any_unit.symbol : nrf_unit_symbol_identifier :=
    nrf_derive_derived_unit_symbol(SELF);
WHERE
  has_valid_elements: (SIZEOF(elements) > 1) OR
    ((SIZEOF(elements) = 1) AND (elements[1].exponent <> 1.0));
  has_valid_exponents: nrf_verify_dimensional_exponents_for_derived_unit(SELF);
END_ENTITY;
```

Attribute definitions:

- elements specifies the expression of units with exponents that defines the derived unit.
- symbol is the derived string of symbols and exponents from the defining elements.

Formal propositions:

- has_valid_elements: The derived unit shall be defined by more than one nrf_derived_unit_element or it shall be defined by one nrf_derived_unit_element with an exponent not equal to one.
- has_valid_exponents: The dimensional exponents as specified in the quantity_category of the **nrf_derived_unit** shall be consistent with the dimensional exponents computed from its constituing unit elements.

4.2.4.10 ENTITY nrf_derived_unit_element

An **nrf_derived_unit_element** is the combination of an nrf_any_unit with an exponent.

NOTE This entity is used to represent an element of an nrf_derived_unit.

Express specification:

```
ENTITY nrf_derived_unit_element;
  unit     : nrf_base_unit;
  exponent : REAL;
END_ENTITY;
```


Attribute definitions:

- unit specifies the nrf_base_unit that is the unit of the element.
- exponent specifies the power that is applied to the unit attribute.

4.2.4.11 FUNCTION nrf_verify_dimensional_exponents

The convenience function **nrf_verify_dimensional_exponents** verifies that the exponents of an nrf_physical_quantity_category are equal to a given set of seven SI dimensional exponent parameters. The function returns TRUE when all corresponding dimensional exponents are equal and returns FALSE otherwise.

Express specification:

```
FUNCTION nrf_verify_dimensional_exponents(
  a_pqc : nrf_physical_quantity_category;
  length_exponent : REAL;
  mass_exponent : REAL;
  time_exponent : REAL;
  electric_current_exponent : REAL;
  thermodynamic_temperature_exponent : REAL;
  amount_of_substance_exponent : REAL;
  luminous_intensity_exponent : REAL) : BOOLEAN;

  IF (a_pqc.length_exponent <> length_exponent)THEN
    RETURN(FALSE);
  END_IF;
  IF (a_pqc.mass_exponent <> mass_exponent) THEN
    RETURN(FALSE);
  END_IF;
  IF (a_pqc.time_exponent <> time_exponent) THEN
    RETURN(FALSE);
  END_IF;
  IF (a_pqc.electric_current_exponent <> electric_current_exponent) THEN
    RETURN(FALSE);
  END_IF;
  IF (a_pqc.thermodynamic_temperature_exponent <>
      thermodynamic_temperature_exponent) THEN
    RETURN(FALSE);
  END_IF;
  IF (a_pqc.amount_of_substance_exponent <> amount_of_substance_exponent) THEN
    RETURN(FALSE);
  END_IF;
  IF (a_pqc.luminous_intensity_exponent <> luminous_intensity_exponent) THEN
    RETURN(FALSE);
  END_IF;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- a_pqc specifies the nrf_physical_quantity_category for which the dimensional exponent values need to be verified
- length_exponent specifies the required length exponent value
- mass_exponent specifies the required mass exponent value
- time_exponent specifies the required time exponent value
- electric_current_exponent specifies the required electric current exponent value

- thermodynamic_temperature_exponent specifies the required thermodynamic temperature exponent value
- amount_of_substance_exponent specifies the required amount of substance exponent value
- luminous_intensity_exponent specifies the required luminous intensity exponent value

4.2.4.12 FUNCTION nrf_verify_equal_dimensional_exponents_for_quantity_categories

The convenience function **nrf_verify_equal_dimensional_exponents_for_quantity_categories** verifies that the dimensional exponents for two nrf_physical_quantity_category instances are equal. The function returns TRUE when all corresponding dimensional exponents are equal and returns FALSE otherwise.

Express specification:

```
FUNCTION nrf_verify_equal_dimensional_exponents_for_quantity_categories(
  a_pqc1 : nrf_physical_quantity_category;
  a_pqc2 : nrf_physical_quantity_category) : BOOLEAN;

  IF (a_pqc1.length_exponent <>
      a_pqc2.length_exponent) THEN
    RETURN(FALSE);
  END_IF;
  IF (a_pqc1.mass_exponent <>
      a_pqc2.mass_exponent) THEN
    RETURN(FALSE);
  END_IF;
  IF (a_pqc1.time_exponent <>
      a_pqc2.time_exponent) THEN
    RETURN(FALSE);
  END_IF;
  IF (a_pqc1.electric_current_exponent <>
      a_pqc2.electric_current_exponent) THEN
    RETURN(FALSE);
  END_IF;
  IF (a_pqc1.thermodynamic_temperature_exponent <>
      a_pqc2.thermodynamic_temperature_exponent) THEN
    RETURN(FALSE);
  END_IF;
  IF (a_pqc1.amount_of_substance_exponent <>
      a_pqc2.amount_of_substance_exponent) THEN
    RETURN(FALSE);
  END_IF;
  IF (a_pqc1.luminous_intensity_exponent <>
      a_pqc2.luminous_intensity_exponent) THEN
    RETURN(FALSE);
  END_IF;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- a_pqc1 specifies the first nrf_physical_quantity_category for which the dimensional exponents need to be verified
- a_pqc2 specifies the second nrf_physical_quantity_category for which the dimensional exponents need to be verified

4.2.4.13 FUNCTION nrf_verify_base_name_and_exponents_for_extended_si_unit

The function **nrf_verify_base_name_and_exponents_for_extended_si_unit** verifies whether a valid base name and the correct dimensional exponents are specified for a given (extended) SI unit. The function returns TRUE if this is the case and FALSE otherwise.

Express specification:

```
FUNCTION nrf_verify_base_name_and_exponents_for_extended_si_unit(
    an_extended_si_unit : nrf_extended_si_unit) : BOOLEAN;
  LOCAL
    ppqc : nrf_primary_physical_quantity_category;
  END_LOCAL;
  ppqc := an_extended_si_unit.quantity_category;

  CASE an_extended_si_unit.base_name OF
    'one'             : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0));
    'metre'           : RETURN(nrf_verify_dimensional_exponents(ppqc,
      1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0));
    'gram'            : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0));
    'second'          : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0));
    'ampere'          : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0));
    'kelvin'          : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0));
    'mole'            : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0));
    'candela'         : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0));
    'radian'          : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0));
    'steradian'       : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0));
    'hertz'           : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 0.0,-1.0, 0.0, 0.0, 0.0, 0.0));
    'newton'          : RETURN(nrf_verify_dimensional_exponents(ppqc,
      1.0, 1.0,-2.0, 0.0, 0.0, 0.0, 0.0));
    'pascal'          : RETURN(nrf_verify_dimensional_exponents(ppqc,
      -1.0, 1.0,-2.0, 0.0, 0.0, 0.0, 0.0));
    'joule'           : RETURN(nrf_verify_dimensional_exponents(ppqc,
      2.0, 1.0,-2.0, 0.0, 0.0, 0.0, 0.0));
    'watt'            : RETURN(nrf_verify_dimensional_exponents(ppqc,
      2.0, 1.0,-3.0, 0.0, 0.0, 0.0, 0.0));
    'coulomb'         : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0));
    'volt'            : RETURN(nrf_verify_dimensional_exponents(ppqc,
      2.0, 1.0,-3.0,-1.0, 0.0, 0.0, 0.0));
    'farad'           : RETURN(nrf_verify_dimensional_exponents(ppqc,
      -2.0,-1.0, 4.0, 2.0, 0.0, 0.0, 0.0));
    'ohm'             : RETURN(nrf_verify_dimensional_exponents(ppqc,
      2.0, 1.0,-3.0,-2.0, 0.0, 0.0, 0.0));
    'siemens'         : RETURN(nrf_verify_dimensional_exponents(ppqc,
      -2.0,-1.0, 3.0, 2.0, 0.0, 0.0, 0.0));
    'weber'           : RETURN(nrf_verify_dimensional_exponents(ppqc,
      2.0, 1.0,-2.0,-1.0, 0.0, 0.0, 0.0));
    'tesla'           : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 1.0,-2.0,-1.0, 0.0, 0.0, 0.0));
    'henry'           : RETURN(nrf_verify_dimensional_exponents(ppqc,
      2.0, 1.0,-2.0,-2.0, 0.0, 0.0, 0.0));
```

```
    'degree_Celsius' : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0));
    'lumen'          : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0));
    'lux'            : RETURN(nrf_verify_dimensional_exponents(ppqc,
      -2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0));
    'becquerel'      : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 0.0,-1.0, 0.0, 0.0, 0.0, 0.0));
    'gray'           : RETURN(nrf_verify_dimensional_exponents(ppqc,
      2.0, 0.0,-2.0, 0.0, 0.0, 0.0, 0.0));
    'sievert'        : RETURN(nrf_verify_dimensional_exponents(ppqc,
      2.0, 0.0,-2.0, 0.0, 0.0, 0.0, 0.0));
    'bit'            : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 0.0,0.0, 0.0, 0.0, 0.0, 0.0));
    'byte'           : RETURN(nrf_verify_dimensional_exponents(ppqc,
      0.0, 0.0,0.0, 0.0, 0.0, 0.0, 0.0));
    OTHERWISE        : RETURN(FALSE);
  END_CASE;
END_FUNCTION;
```

Argument definitions:

— an_extended_si_unit specifies the nrf_extended_si_unit to be verified.

4.2.4.14 FUNCTION nrf_derive_extended_si_symbol

The function **nrf_derive_extended_si_symbol** returns the symbol string for a given SI unit by concatenatig the SI prefix symbol and SI base_name symbol.

Express specification:

```
FUNCTION nrf_derive_extended_si_symbol(
    an_extended_si_unit : nrf_extended_si_unit) : nrf_unit_symbol_identifier;
  LOCAL
    symbol : STRING := '';
    result : nrf_unit_symbol_identifier;
  END_LOCAL;
  CASE an_extended_si_unit.prefix OF
    ''        : symbol := '';
    'yotta'   : symbol := 'Y';
    'zetta'   : symbol := 'Z';
    'exa'     : symbol := 'E';
    'peta'    : symbol := 'P';
    'tera'    : symbol := 'T';
    'giga'    : symbol := 'G';
    'mega'    : symbol := 'M';
    'kilo'    : symbol := 'k';
    'hecto'   : symbol := 'h';
    'deca'    : symbol := 'da';
    'deci'    : symbol := 'd';
    'centi'   : symbol := 'c';
    'milli'   : symbol := 'm';
    'micro'   : symbol := 'micro';
    'nano'    : symbol := 'n';
    'pico'    : symbol := 'p';
    'femto'   : symbol := 'f';
    'atto'    : symbol := 'a';
    'zepto'   : symbol := 'z';
    'yocto'   : symbol := 'y';
    'exbi'    : symbol := 'Ei';
    'pebi'    : symbol := 'Pi';
```



```
    'tebi'    : symbol := 'Ti';
    'gibi'    : symbol := 'Gi';
    'mebi'    : symbol := 'Mi';
    'kibi'    : symbol := 'Ki';
    OTHERWISE : symbol := '?';
  END_CASE;
  CASE an_extended_si_unit.base_name OF
    'one'            : symbol := '-';
    'metre'          : symbol := symbol + 'm';
    'gram'           : symbol := symbol + 'g';
    'second'         : symbol := symbol + 's';
    'ampere'         : symbol := symbol + 'A';
    'kelvin'         : symbol := symbol + 'K';
    'mole'           : symbol := symbol + 'mol';
    'candela'        : symbol := symbol + 'cd';
    'radian'         : symbol := symbol + 'rad';
    'steradian'      : symbol := symbol + 'sr';
    'hertz'          : symbol := symbol + 'Hz';
    'newton'         : symbol := symbol + 'N';
    'pascal'         : symbol := symbol + 'Pa';
    'joule'          : symbol := symbol + 'J';
    'watt'           : symbol := symbol + 'W';
    'coulomb'        : symbol := symbol + 'C';
    'volt'           : symbol := symbol + 'V';
    'farad'          : symbol := symbol + 'F';
    'ohm'            : symbol := symbol + 'Ohm';
    'siemens'        : symbol := symbol + 'S';
    'weber'          : symbol := symbol + 'Wb';
    'tesla'          : symbol := symbol + 'T';
    'henry'          : symbol := symbol + 'H';
    'degree_Celsius' : symbol := symbol + 'degC';
    'lumen'          : symbol := symbol + 'lm';
    'lux'            : symbol := symbol + 'lux';
    'becquerel'      : symbol := symbol + 'Bq';
    'gray'           : symbol := symbol + 'Gy';
    'sievert'        : symbol := symbol + 'Sv';
    'bit'            : symbol := symbol + 'b';
    'byte'           : symbol := symbol + 'B';
    OTHERWISE        : symbol := symbol + '?';
  END_CASE;
  result := symbol;
  RETURN(result);
END_FUNCTION;
```

Argument definitions:

— an_extended_si_unit specifies the (extended) SI unit for which to derive the symbol

4.2.4.15 FUNCTION nrf_derive_derived_unit_symbol

The **nrf_derive_derived_unit_symbol** function returns a concatenated symbol string for the expression of unit and exponent elements in a given nrf_derived_unit.

Express specification:

```
FUNCTION nrf_derive_derived_unit_symbol(
    a_derived_unit : nrf_derived_unit) : nrf_unit_symbol_identifier;
  LOCAL
    i : INTEGER;
    numerator_symbol : STRING := '';
    n_numerator_symbols : INTEGER := 0;
```



```
  denominator_symbol : STRING := '';
  n_denominator_symbols : INTEGER := 0;
  exponent_string : STRING;
  format_exponent_string : STRING;
  abs_exponent : REAL;
  result : nrf_unit_symbol_identifier;
  is_nested_derived_unit : LOGICAL;
END_LOCAL;
REPEAT i := LOINDEX(a_derived_unit.elements) TO HIINDEX(a_derived_unit.elements);
  abs_exponent := ABS(a_derived_unit.elements[i].exponent);
  IF (abs_exponent = 1.0) THEN
    exponent_string := '';
  ELSE
    exponent_string := '^';
    CASE abs_exponent OF
      2.0: exponent_string := exponent_string + '2';
      3.0: exponent_string := exponent_string + '3';
      4.0: exponent_string := exponent_string + '4';
      5.0: exponent_string := exponent_string + '5';
      6.0: exponent_string := exponent_string + '6';
      7.0: exponent_string := exponent_string + '7';
      8.0: exponent_string := exponent_string + '8';
      9.0: exponent_string := exponent_string + '9';
      1.0/2.0: exponent_string := exponent_string + '(1/2)';
      1.0/3.0: exponent_string := exponent_string + '(1/3)';
      1.0/4.0: exponent_string := exponent_string + '(1/4)';
      1.0/5.0: exponent_string := exponent_string + '(1/5)';
      1.0/6.0: exponent_string := exponent_string + '(1/6)';
      1.0/7.0: exponent_string := exponent_string + '(1/7)';
      1.0/8.0: exponent_string := exponent_string + '(1/8)';
      1.0/9.0: exponent_string := exponent_string + '(1/9)';
      OTHERWISE:
        BEGIN
          format_exponent_string := FORMAT(abs_exponent, '+5.2F');
          exponent_string := exponent_string +
            format_exponent_string[2:LENGTH(format_exponent_string)];
        END;
    END_CASE;
  END_IF;
  is_nested_derived_unit :=
    'NRF_ARM.NRF_DERIVED_UNIT' IN TYPEOF(a_derived_unit.elements[i]);
  IF (a_derived_unit.elements[i].exponent >= 0.0) THEN
    n_numerator_symbols := n_numerator_symbols + 1;
    IF n_numerator_symbols > 1 THEN
      numerator_symbol := numerator_symbol + '.';
    END_IF;
    IF is_nested_derived_unit THEN
      numerator_symbol := numerator_symbol +
        '(' + a_derived_unit.elements[i].unit.symbol + ')' + exponent_string;
    ELSE
      numerator_symbol := numerator_symbol +
        a_derived_unit.elements[i].unit.symbol + exponent_string;
    END_IF;
  ELSE
    n_denominator_symbols := n_denominator_symbols + 1;
    IF n_denominator_symbols > 1 THEN
      denominator_symbol := denominator_symbol + '.';
    END_IF;
    IF is_nested_derived_unit THEN
      denominator_symbol := denominator_symbol +
        '(' + a_derived_unit.elements[i].unit.symbol + ')' + exponent_string;
    ELSE
      denominator_symbol := denominator_symbol +
        a_derived_unit.elements[i].unit.symbol + exponent_string;
```

```
      END_IF;
    END_IF;
  END_REPEAT;
  IF n_denominator_symbols = 0 THEN
    result := numerator_symbol;
  ELSE
    IF n_numerator_symbols = 0 THEN
      IF n_denominator_symbols = 1 THEN
        result := denominator_symbol + '^-1';
      ELSE
        result := '(' + denominator_symbol + ')^-1';
      END_IF;
    ELSE
      IF n_denominator_symbols = 1 THEN
        result := numerator_symbol + '/' + denominator_symbol;
      ELSE
        result := numerator_symbol + '/(' + denominator_symbol + ')';
      END_IF;
    END_IF;
  END_IF;
  RETURN(result);
END_FUNCTION;
```

Argument definitions:

— a_derived_unit specifies the name of the derived unit

4.2.4.16 FUNCTION nrf_verify_dimensional_exponents_for_derived_unit

The function **nrf_verify_dimensional_exponents_for_derived_unit** verifies that the dimensional exponents as specified in the quantity_category of an nrf_derived_unit are consistent with the dimensional exponents computed from its constituing unit elements.

Express specification:

```
FUNCTION nrf_verify_dimensional_exponents_for_derived_unit(
    a_derived_unit : nrf_derived_unit) : BOOLEAN;
  LOCAL
    length_exponent                    : REAL := 0.0;
    mass_exponent                      : REAL := 0.0;
    time_exponent                      : REAL := 0.0;
    electric_current_exponent          : REAL := 0.0;
    thermodynamic_temperature_exponent : REAL := 0.0;
    amount_of_substance_exponent       : REAL := 0.0;
    luminous_intensity_exponent        : REAL := 0.0;
    element_qc                         : nrf_primary_physical_quantity_category;
    element_exp                        : REAL;
  END_LOCAL;
  REPEAT i := 1 TO SIZEOF(a_derived_unit.elements);
    element_qc := a_derived_unit.elements[i].unit.quantity_category;
    element_exp := a_derived_unit.elements[i].exponent;
    length_exponent :=
      length_exponent +
      element_qc.length_exponent * element_exp;
    mass_exponent :=
      mass_exponent +
      element_qc.mass_exponent * element_exp;
    time_exponent :=
      time_exponent +
      element_qc.time_exponent * element_exp;
    electric_current_exponent :=
```



```
      electric_current_exponent +
      element_qc.electric_current_exponent * element_exp;
    thermodynamic_temperature_exponent :=
      thermodynamic_temperature_exponent +
      element_qc.thermodynamic_temperature_exponent * element_exp;
    amount_of_substance_exponent :=
      amount_of_substance_exponent +
      element_qc.amount_of_substance_exponent * element_exp;
    luminous_intensity_exponent :=
      luminous_intensity_exponent +
      element_qc.luminous_intensity_exponent * element_exp;
  END_REPEAT;
  RETURN(nrf_verify_dimensional_exponents(
    a_derived_unit.quantity_category,
    length_exponent,
    mass_exponent,
    time_exponent,
    electric_current_exponent,
    thermodynamic_temperature_exponent,
    amount_of_substance_exponent,
    luminous_intensity_exponent));
END_FUNCTION;
```

Argument definitions:

— a_derived_unit specifies the nrf_derived_unit to be verified.

4.2.4.17 FUNCTION nrf_derive_extended_si_prefix_factor

The **nrf_derive_extended_si_prefix_factor** function returns the decimal multiplication factor for a given SI or binary prefix string.

Express specification:

```
FUNCTION nrf_derive_extended_si_prefix_factor(
    an_extended_si_prefix : nrf_label) : REAL;
  LOCAL
    factor : REAL;
  END_LOCAL;
  CASE an_extended_si_prefix OF
    'yotta'   : factor := 1.0E24;
    'zetta'   : factor := 1.0E21;
    'exa'     : factor := 1.0E18;
    'peta'    : factor := 1.0E15;
    'tera'    : factor := 1.0E12;
    'giga'    : factor := 1.0E9;
    'mega'    : factor := 1.0E06;
    'kilo'    : factor := 1000.0;
    'hecto'   : factor := 100.0;
    'deca'    : factor := 10.0;
    ''        : factor := 1.0;
    'deci'    : factor := 0.1;
    'centi'   : factor := 0.01;
    'milli'   : factor := 0.001;
    'micro'   : factor := 1.0E-06;
    'nano'    : factor := 1.0E-09;
    'pico'    : factor := 1.0E-12;
    'femto'   : factor := 1.0E-15;
    'atto'    : factor := 1.0E-18;
    'zepto'   : factor := 1.0E-21;
    'yocto'   : factor := 1.0E-24;
```



```
    'exbi'    : factor := 2.0**60;
    'pebi'    : factor := 2.0**50;
    'tebi'    : factor := 2.0**40;
    'gibi'    : factor := 2.0**30;
    'mebi'    : factor := 2.0**20;
    'kibi'    : factor := 2.0**10;
   OTHERWISE : factor := 1.0;
  END_CASE;
  RETURN(factor);
 END_FUNCTION;
```

Argument definitions:

— an_extended_si_prefix specifies the SI or binary prefix for which the multiplication factor needs to be returned

4.2.4.18 TYPE nrf_uncertainty_margins_type

The **nrf_uncertainty_margins_type** type specifies an enumeration of the possible ways to specify uncertainty margins for a physical quantity.

Express specification:

```
TYPE nrf_uncertainty_margins_type = ENUMERATION OF(
  SYMMETRIC_ABSOLUTE,
  SYMMETRIC_PERCENTAGE,
  ASYMMETRIC_ABSOLUTE,
  ASYMMETRIC_PERCENTAGE);
END_TYPE;
```

Enumeration value definitions:

— SYMMETRIC_ABSOLUTE specifies that the uncertainty is defined by one positive absolute symmetrical margin value. In human readable notation a corresponding quantity value would be written as follows:
$x \pm y$
where: x is the best estimated value and y the uncertainty margin value.

— SYMMETRIC_PERCENTAGE specifies that the uncertainty margin is defined by one relative symmetrical margin percentage. In human readable notation a corresponding quantity value would be written as follows:
$x \pm y\,\%$
where: x is the best estimated value and y the uncertainty margin percentage.

— ASYMMETRIC_ABSOLUTE specifies that the uncertainty margins are defined by a positive upper absolute margin value and a negative lower absolute margin value. In human readable notation a corresponding quantity value would be written as follows:
$x\,^{+y_{upper}}_{-y_{lower}}$
where: x is the best estimated value and y_{upper} and y_{lower} are the upper and lower margin values.

— ASYMMETRIC_PERCENTAGE specifies that the uncertainty margins are defined by a positive upper margin percentage and a negative lower margin percentage. In human readable notation a corresponding quantity value would be written as follows:
$x\,^{+y_{upper}\,\%}_{-y_{lower}\,\%}$
where: x is the best estimated value and y_{upper} and y_{lower} are the upper and lower margin percentages.

4.2.4.19 ENTITY nrf_uncertainty_probability_distribution

An **nrf_uncertainty_probability_distribution** specifies a probability density distribution function for the uncertainty in values for a quantity and how to interpret the uncertainty margin(s).

EXAMPLE An example of an **nrf_uncertainty_probability_distribution** is the *normal distribution* with name "Normal", description "Normal or Gaussian probability density distribution, see e.g. http://en.wikipedia.org/wiki/Normal_Distribution. The best estimated value is the mean and the margin value is the standard deviation.". The valid_uncertainty_margins_types would contain SYMMETRIC_ABSOLUTE and SYMMETRIC_PERCENTAGE.

Express specification:

```
ENTITY nrf_uncertainty_probability_distribution;
  name                           : nrf_non_blank_label;
  description                    : nrf_text;
  valid_uncertainty_margins_types : SET OF nrf_uncertainty_margins_type;
UNIQUE
  has_unique_name: name;
END_ENTITY;
```

Attribute definitions:

- name specifies the name of a probability density distribution function.
- description specifies a textual description of the distribution.
- valid_uncertainty_margins_types specifies the applicable uncertainty margin types for quantities adhering to the distribution.

Formal propositions:

- has_unique_name: The name shall be unique in the dataset.

4.2.4.20 ENTITY nrf_uncertainty_specification_method

An **nrf_uncertainty_specification_method** specifies the way in which uncertainty for a physical quantity is handled.

Express specification:

```
ENTITY nrf_uncertainty_specification_method;
  margins      : nrf_uncertainty_margins_type;
  distribution : OPTIONAL nrf_uncertainty_probability_distribution;
UNIQUE
  has_unique_tuple_margins_distribution: margins, distribution;
END_ENTITY;
```

Attribute definitions:

- margins specifies what combination of absolute or percentage, and symmetric or asymmetric, margin values apply.
- distribution optionally specifies what probability distribution applies to the uncertainty.

Formal propositions:

— has_unique_tuple_margins_distribution: The tuple (margins, distribution) shall be unique in the dataset.

4.2.4.21 ENTITY nrf_physical_quantity_category

An **nrf_physical_quantity_category** is an abstract supertype that provides a generic mechanism for referencing nrf_basic_physical_quantity_category and nrf_qualified_physical_quantity_category instances. A physical quantity category captures the essential common characteristics of a collection of scalar physical quantities of the same kind. It specifies the physical dimension of a quantity through dimensional exponents for each of the seven base quantities of the SI unit system as defined in [ISO 31]. All physical quantities are founded on these seven base quantities. For dimensionless quantities and for quantities that cannot be related to the SI quantities and units, all seven dimensional exponents shall be set to zero. In addition it is possible to specify whether and in what way uncertainty in the values for a given quantity category shall be handled.

Express specification:

```
ENTITY nrf_physical_quantity_category
  ABSTRACT SUPERTYPE OF(ONEOF(
    nrf_basic_physical_quantity_category,
    nrf_qualified_physical_quantity_category));
  name                                : nrf_non_blank_label;
  symbol                              : nrf_non_blank_label;
  description                         : nrf_text;
  length_exponent                     : REAL;
  mass_exponent                       : REAL;
  time_exponent                       : REAL;
  electric_current_exponent           : REAL;
  thermodynamic_temperature_exponent  : REAL;
  amount_of_substance_exponent        : REAL;
  luminous_intensity_exponent         : REAL;
UNIQUE
  has_unique_name: name;
END_ENTITY;
```

Attribute definitions:

— name specifies the human-interpretable name of an **nrf_physical_quantity_category**.

— symbol specifies the alternative short name of an **nrf_physical_quantity_category**, which can be used to identify it when brevity is needed, as for example in a mathematical equation, a table heading or a programming language.

— description specifies the textual description of an **nrf_physical_quantity_category**.

— length_exponent specifies the power of the length dimension of the quantity category.

— mass_exponent specifies the power of the mass dimension of the quantity category.

— time_exponent specifies the power of the time dimension of the quantity category.

— electric_current_exponent specifies the power of the electric current dimension of the quantity category.

— thermodynamic_temperature_exponent specifies the power of the thermodynamic temperature dimension of the quantity category.

— amount_of_substance_exponent specifies the power of the amount of substance dimension of the quantity category.

— luminous_intensity_exponent specifies the power of the luminous intensity dimension of the quantity category.

Formal propositions:

— has_unique_name: The name shall be unique in the dataset.

4.2.4.22 ENTITY nrf_basic_physical_quantity_category

An **nrf_basic_physical_quantity_category** specifies an abstract supertype that provides a generic mechanism to reference an nrf_primary_physical_quantity_category or an nrf_secondary_physical_quantity_category. This is needed to enable the definition of nrf_qualified_physical_quantity_category instances for both primary and secondary physical quantity categories.

Express specification:

```
ENTITY nrf_basic_physical_quantity_category
  ABSTRACT SUPERTYPE OF(ONEOF(
   nrf_primary_physical_quantity_category,
   nrf_secondary_physical_quantity_category))
  SUBTYPE OF(nrf_physical_quantity_category);
END_ENTITY;
```

4.2.4.23 ENTITY nrf_primary_physical_quantity_category

An **nrf_primary_physical_quantity_category** is a type of nrf_basic_physical_quantity_category that specifies a primary scalar physical quantity category. The distinction between a primary and a secondary physical quantity category is that in case of alternative names for a specific quantity category the primary category is the one with the more generic, more frequently used name and the secondary category is the one with the more specialized, less frequently used name.

EXAMPLE 1 Examples of primary physical quantity categories are the seven SI base quantities: *length*, *mass*, *time*, *electric current* , *thermodynamic temperature*, *amount of substance* and *luminous intensity*, and the 21 SI derived quantities with special names: *energy*, *power*, *force*, etc.

EXAMPLE 2 Examples of secondary physical quantity categories that are associated with the primary category *length* are: *diameter*, *radius*, *distance*, *width*, *depth* and *wavelength*.

Express specification:

```
ENTITY nrf_primary_physical_quantity_category
  SUBTYPE OF(nrf_basic_physical_quantity_category);
END_ENTITY;
```

4.2.4.24 ENTITY nrf_secondary_physical_quantity_category

An **nrf_secondary_physical_quantity_category** is a type of nrf_basic_physical_quantity_category that specifies an alternative scalar physical quantity category of the same kind and associated with an nrf_primary_physical_quantity_category. The secondary quantity category is a less frequently used name or a specialization of the primary category. A secondary quantity category always has exactly the same physical dimensions as its associated primary category.

EXAMPLE Examples of secondary physical quantity categories that are associated with the primary category *power* are: *heat* and *dissipation*.

Express specification:

```
ENTITY nrf_secondary_physical_quantity_category
  SUBTYPE OF(nrf_basic_physical_quantity_category);
  primary_category : nrf_primary_physical_quantity_category;
DERIVE
  SELF\nrf_physical_quantity_category.length_exponent : REAL
    := primary_category.length_exponent;
  SELF\nrf_physical_quantity_category.mass_exponent : REAL
    := primary_category.mass_exponent;
  SELF\nrf_physical_quantity_category.time_exponent : REAL
    := primary_category.time_exponent;
  SELF\nrf_physical_quantity_category.electric_current_exponent : REAL
    := primary_category.electric_current_exponent;
  SELF\nrf_physical_quantity_category.thermodynamic_temperature_exponent : REAL
    := primary_category.thermodynamic_temperature_exponent;
  SELF\nrf_physical_quantity_category.amount_of_substance_exponent : REAL
    := primary_category.amount_of_substance_exponent;
  SELF\nrf_physical_quantity_category.luminous_intensity_exponent : REAL
    := primary_category.luminous_intensity_exponent;
END_ENTITY;
```

Attribute definitions:

- primary_category specifies the associated nrf_primary_physical_quantity_category.
- length_exponent specifies the power of the length dimension of the quantity category, which is by definition equal to that of the primary_category.
- mass_exponent specifies the power of the mass dimension of the quantity category, which is by definition equal to that of the primary_category.
- time_exponent specifies the power of the time dimension of the quantity category, which is by definition equal to that of the primary_category.
- electric_current_exponent specifies the power of the electric current dimension of the quantity category, which is by definition equal to that of the primary_category.
- thermodynamic_temperature_exponent specifies the power of the thermodynamic temperature dimension of the quantity category, which is by definition the same as that of the primary_category.
- amount_of_substance_exponent specifies the power of the amount of substance dimension of the quantity category, which is by definition equal to that of the primary_category.
- luminous_intensity_exponent specifies the power of the luminous intensity dimension of the quantity category, which is by definition equal to that of the primary_category.

4.2.4.25 ENTITY nrf_qualified_physical_quantity_category

An **nrf_qualified_physical_quantity_category** is a type of nrf_physical_quantity_category that specifies a qualified primary or secondary physical quantity category. Such a quantity is qualified by attaching one or more quantity qualifiers that specialize the usage of the quantity category.

EXAMPLE 1 Examples of qualified primary physical quantity categories are: *minimum length*, *switch_on temperature* and *maximum power*.

EXAMPLE 2 Examples of qualified secondary physical quantity categories are: *maximum width* and *lower_limit dissipation*.

Express specification:


```
ENTITY nrf_qualified_physical_quantity_category
  SUBTYPE OF(nrf_physical_quantity_category);
  basic_category : nrf_basic_physical_quantity_category;
  qualifiers     : LIST [1:?] OF UNIQUE nrf_quantity_qualifier;
DERIVE
  SELF\nrf_physical_quantity_category.name : nrf_non_blank_label
    := nrf_get_qualified_quantity_name(SELF);
  SELF\nrf_physical_quantity_category.symbol : nrf_non_blank_label
    := nrf_get_qualified_quantity_symbol(SELF);
  SELF\nrf_physical_quantity_category.description : nrf_text
    := nrf_get_qualified_quantity_description(SELF);
  SELF\nrf_physical_quantity_category.length_exponent : REAL
    := basic_category.length_exponent;
  SELF\nrf_physical_quantity_category.mass_exponent : REAL
    := basic_category.mass_exponent;
  SELF\nrf_physical_quantity_category.time_exponent : REAL
    := basic_category.time_exponent;
  SELF\nrf_physical_quantity_category.electric_current_exponent : REAL
    := basic_category.electric_current_exponent;
  SELF\nrf_physical_quantity_category.thermodynamic_temperature_exponent : REAL
    := basic_category.thermodynamic_temperature_exponent;
  SELF\nrf_physical_quantity_category.amount_of_substance_exponent : REAL
    := basic_category.amount_of_substance_exponent;
  SELF\nrf_physical_quantity_category.luminous_intensity_exponent : REAL
    := basic_category.luminous_intensity_exponent;
END_ENTITY;
```

Attribute definitions:

- basic_category specifies the associated nrf_basic_physical_quantity_category.
- name is the derived fully qualified name of an **nrf_qualified_physical_quantity_category**, that consists of the concatenation of the names of the qualifiers followed by the name of the basic_category, joined by underscores.
- symbol is the derived fully qualified symbol of an **nrf_qualified_physical_quantity_category**, that consists of the symbol of the basic_category followed by the symbols of the qualifiers in reverse order, joined by underscores, thereby mimicing mathematical subscript notation.
- description is the derived fully qualified description of an **nrf_qualified_physical_quantity_category**, that consists of the description of the basic_category followed by the descriptions of the qualifiers.
- length_exponent specifies the power of the length dimension of the quantity category, which is by definition equal to that of the basic_category.
- mass_exponent specifies the power of the mass dimension of the quantity category, which is by definition equal to that of the basic_category.
- time_exponent specifies the power of the time dimension of the quantity category, which is by definition equal to that of the basic_category.
- electric_current_exponent specifies the power of the electric current dimension of the quantity category, which is by definition equal to that of the basic_category.
- thermodynamic_temperature_exponent specifies the power of the thermodynamic temperature dimension of the quantity category, which is by definition the same as that of the basic_category.
- amount_of_substance_exponent specifies the power of the amount of substance dimension of the quantity category, which is by definition equal to that of the basic_category.
- luminous_intensity_exponent specifies the power of the luminous intensity dimension of the quantity category, which is by definition equal to that of the basic_category.

4.2.4.26 ENTITY nrf_quantity_qualifier

An **nrf_quantity_qualifier** specifies a qualification for an nrf_basic_physical_quantity_category. One or more qualifier names further specialize the meaning and usage of an nrf_primary_physical_quantity_category or nrf_secondary_physical_quantity_category over its basic meaning and usage.

EXAMPLE 1 Typical quantity qualifier names that could be used with many different primary and secondary quantity categories are: *minimum*, *maximum*, *average*, *mean*, *static*, *dynamic*, *lower_limit*, *upper_limit*, *switch_on* , *switch_off*, etc.

EXAMPLE 2 Quantity qualifier names that could be used with thermo-optical quantities like *reflectance* are: *solar*, *infra_red*, *specular* and *diffuse*.

Express specification:

```
ENTITY nrf_quantity_qualifier;
  name        : nrf_non_blank_label;
  symbol      : nrf_non_blank_label;
  description : nrf_text;
UNIQUE
  has_unique_name: name;
END_ENTITY;
```

Attribute definitions:

- name specifies the human-interpretable name of an **nrf_quantity_qualifier**.
- symbol specifies the alternative short name of an **nrf_quantity_qualifier**, which can be used to identify it when brevity is needed as for example in an equation, a table heading or a programming language.
- description specifies the textual description of an **nrf_quantity_qualifier**.

Formal propositions:

- has_unique_name: The name shall be unique in the dataset.

4.2.4.27 ENTITY nrf_any_quantity_type

The **nrf_any_quantity_type** is an abstract supertype that provides a generic mechanism to reference an nrf_any_scalar_quantity_type, an nrf_any_tensor_quantity_type or any of their subtypes. A quantity type is defined by a name, symbol, description and rank, and the datatype(s) used to store the quantity values. For tensor quantity types it also specifies the dimensions and quantity types for each of the tensor's elements. Physical quantity types may also specify an uncertainty method that captures how uncertainty margins and uncertainty probability distribution are treated. The table below specifies the number of real values and the number of integer values to be held for quantities typed by any of the possible quantity types. The possible quantity types in the first column are the leaf subtypes of **nrf_any_quantity_type**.

quantity type	uncertainty_method.margins	number_of_real_values	number_of_integer_values
nrf_real_quantity_type	not specified	1	0
nrf_real_quantity_type	SYMMETRIC_ABSOLUTE	2	0

quantity type	uncertainty_method.margins	number_of_real_values	number_of_integer_values
nrf_real_quantity_type	SYMMETRIC_PERCENTAGE	2	0
nrf_real_quantity_type	ASYMMETRIC_ABSOLUTE	3	0
nrf_real_quantity_type	ASYMMETRIC_PERCENTAGE	3	0
nrf_integer_quantity_type	not specified	0	1
nrf_integer_quantity_type	SYMMETRIC_ABSOLUTE	1	1
nrf_integer_quantity_type	SYMMETRIC_PERCENTAGE	1	1
nrf_integer_quantity_type	ASYMMETRIC_ABSOLUTE	2	1
nrf_integer_quantity_type	ASYMMETRIC_PERCENTAGE	2	1
nrf_enumeration_quantity_type	not applicable	0	1
nrf_string_quantity_type	not applicable	0	1

quantity type	uncertainty_method.margins	number_of_real_values	number_of_integer_values
nrf_general_tensor_quantity_type	specified per element	sum over all elements	sum over all elements
nrf_symmetric_matrix_quantity_type	specified per element	sum over all elements	sum over all elements
nrf_anti_symmetric_matrix_quantity_type	specified per element	sum over all elements	sum over all elements

EXAMPLE 1 Examples of scalar quantity types are "length" in "metre", "radius" in "millimetre", "diameter" in "inch", "infra_red_emittance" in "one" (dimensionless), "static_pressure" in "pascal", "upper_design_temperature" in "degree_Celsius".

EXAMPLE 2 Examples of tensor quantity types are "cartesian_3d_velocity_vector" and "stiffness_matrix".

Express specification:

```
ENTITY nrf_any_quantity_type
  ABSTRACT SUPERTYPE OF(ONEOF(
    nrf_any_scalar_quantity_type,
    nrf_any_tensor_quantity_type));
  name                    : nrf_non_blank_label;
  symbol                  : nrf_non_blank_label;
  description             : nrf_text;
  rank                    : INTEGER;
  number_of_real_values   : INTEGER;
  number_of_integer_values : INTEGER;
END_ENTITY;
```

Attribute definitions:

- name specifies the human-interpretable name of an **nrf_any_quantity_type**.
- symbol specifies the alternative short name of an **nrf_any_quantity_type**, which can be used to identify it when brevity is needed, as for example in a mathematical equation, a table heading or a programming language.
- description specifies the textual description of an **nrf_any_quantity_type**.
- rank specifies the number of dimensions of an **nrf_any_quantity_type**. The rank is zero for a scalar quantity type, one for a vector, two for a matrix, and three or higher for higher order tensors.
- number_of_real_values specifies the total number of real values to be held for a quantity typed by an **nrf_any_quantity_type**. See the table above for details. The number_of_real_values is needed for housekeeping concerned with the indexing in the lists of real_values in nrf_datacube and nrf_any_quantity_value_prescription instances.
- number_of_integer_values specifies the total number of integer values to be held for a quantity typed by an **nrf_any_quantity_type**. See the table above for details. The number_of_integer_values is needed for housekeeping concerned with the indexing in the lists of integer_values in nrf_datacube and nrf_any_quantity_value_prescription instances. Integer values are also used as indices to point to actual enumeration and string values for quantities of nrf_enumeration_quantity_type and nrf_string_quantity_type.

4.2.4.28 ENTITY nrf_any_scalar_quantity_type

The **nrf_any_scalar_quantity_type** is an abstract supertype that provides a generic mechanism to reference an nrf_physical_quantity_type, an nrf_string_quantity_type or an nrf_enumeration_quantity_type and defines the common attributes of all scalar quantity types.

Express specification:

```
ENTITY nrf_any_scalar_quantity_type
  ABSTRACT SUPERTYPE OF(ONEOF(
    nrf_physical_quantity_type,
    nrf_string_quantity_type,
    nrf_enumeration_quantity_type))
  SUBTYPE OF(nrf_any_quantity_type);
DERIVE
  SELF\nrf_any_quantity_type.rank : INTEGER := 0;
END_ENTITY;
```

Attribute definitions:

— rank is derived to be always zero.

4.2.4.29 ENTITY nrf_physical_quantity_type

The **nrf_physical_quantity_type** is an abstract supertype that provides a generic mechanism to reference an nrf_real_quantity_type or an nrf_integer_quantity_type and defines the common attributes of all physical quantity types.

Express specification:

```
ENTITY nrf_physical_quantity_type
  ABSTRACT SUPERTYPE OF(ONEOF(
    nrf_real_quantity_type,
    nrf_integer_quantity_type))
  SUBTYPE OF(nrf_any_scalar_quantity_type);
  quantity_category  : nrf_physical_quantity_category;
  unit               : nrf_any_unit;
  uncertainty_method : OPTIONAL nrf_uncertainty_specification_method;
DERIVE
  SELF\nrf_any_quantity_type.name        : nrf_non_blank_label := quantity_category.name;
  SELF\nrf_any_quantity_type.symbol      : nrf_non_blank_label := quantity_category.symbol;
  SELF\nrf_any_quantity_type.description : nrf_text := quantity_category.description;
UNIQUE
  has_unique_tuple_of_name_unit_uncertainty_method: name, unit, uncertainty_method;
WHERE
  has_same_dimensional_exponents_as_unit:
    nrf_verify_equal_dimensional_exponents_for_quantity_categories( quantity_category,
    unit.quantity_category);
END_ENTITY;
```

Attribute definitions:

— quantity_category specifies the nrf_physical_quantity_category to which this **nrf_physical_quantity_type** belongs.

— unit specifies the (physical) unit in which values for an **nrf_physical_quantity_type** are defined. Where applicable an explicit 'dimensionless' unit shall be specified.

- uncertainty_method optionally specifies how uncertainties in the values of quantities for an **nrf_physical_quantity_type** are expressed. Leaving the uncertainty_method unset implies that no uncertainty for the quantity values is taken into account, therefore only single quantity values will be provided.
- name is derived to be the name of the associated quantity_category.
- symbol is derived to be the symbol of the associated quantity_category.
- description is derived to be the description of the associated quantity_category.

Formal propositions:

- has_unique_tuple_of_name_unit_uncertainty_method: The tuple (name, unit, uncertainty_method) shall be unique.
- has_same_dimensional_exponents_as_unit: The physical quantity type and its unit shall have the same dimensional exponents.

4.2.4.30 ENTITY nrf_real_quantity_type

An **nrf_real_quantity_type** is a type of nrf_physical_quantity_type that specifies a real valued quantity type, i.e. a physical quantity type that captures a magnitude. Optionally a lower bound or an upper bound for a valid range of values may be specified. The number of real values to be held for an **nrf_real_quantity_type** depends on the specified uncertainty_method attribute, as shown in the table in the specification for nrf_any_quantity_type.

Express specification:

```
ENTITY nrf_real_quantity_type
  SUBTYPE OF(nrf_physical_quantity_type);
  lower_bound           : OPTIONAL REAL;
  lower_bound_inclusive : BOOLEAN;
  upper_bound           : OPTIONAL REAL;
  upper_bound_inclusive : BOOLEAN;
DERIVE
  SELF\nrf_any_quantity_type.number_of_real_values : INTEGER :=
    nrf_get_number_of_real_values_for_real_quantity_type(SELF);
  SELF\nrf_any_quantity_type.number_of_integer_values : INTEGER := 0;
WHERE
  has_valid_bounds: (NOT EXISTS(lower_bound)) OR (NOT EXISTS(upper_bound)) OR
    (EXISTS(lower_bound) AND EXISTS(upper_bound) AND (lower_bound <= upper_bound));
END_ENTITY;
```

Attribute definitions:

- lower_bound optionally specifies a lower bound for valid quantity values for an **nrf_real_quantity_type**.
- lower_bound_inclusive specifies whether the lower_bound itself is included or excluded from the valid value range.
- upper_bound optionally specifies an upper bound for valid quantity values for an **nrf_real_quantity_type**.
- upper_bound_inclusive specifies whether the upper_bound itself is included or excluded from the valid value range.
- number_of_real_values is derived to be the number of real quantity values to be held for an **nrf_real_quantity_type** in the real_values list of nrf_datacube or

nrf_any_quantity_value_prescription instances depending on the actual uncertainty_method. It may be 1, 2 or 3.

— number_of_integer_values is derived to be the number of integer quantity values to be held for an **nrf_real_quantity_type** in the integer_values list of nrf_datacube or nrf_any_quantity_value_prescription instances depending on the actual uncertainty_method. It is always zero.

Formal propositions:

— has_valid_bounds: If both lower_bound and upper_bound are specified then the lower_bound shall be less than or equal to the upper_bound.

4.2.4.31 ENTITY nrf_integer_quantity_type

An **nrf_integer_quantity_type** is a type of nrf_physical_quantity_type that specifies an integer valued quantity type, i.e. a physical quantity type that captures a multitude. Optionally a lower bound or an upper bound for a valid range of values may be specified. The number of integer and real values to be held for an **nrf_integer_quantity_type** depends on the specified uncertainty_method attribute, as shown in the table in the specification for nrf_any_quantity_type.

Express specification:

```
ENTITY nrf_integer_quantity_type
  SUBTYPE OF(nrf_physical_quantity_type);
  lower_bound           : OPTIONAL INTEGER;
  lower_bound_inclusive : BOOLEAN;
  upper_bound           : OPTIONAL INTEGER;
  upper_bound_inclusive : BOOLEAN;
DERIVE
  SELF\nrf_any_quantity_type.number_of_real_values : INTEGER :=
    nrf_get_number_of_real_values_for_integer_quantity_type(SELF);
  SELF\nrf_any_quantity_type.number_of_integer_values : INTEGER := 1;
WHERE
  has_valid_bounds: (NOT EXISTS(lower_bound)) OR (NOT EXISTS(upper_bound)) OR
    (EXISTS(lower_bound) AND EXISTS(upper_bound) AND (lower_bound <= upper_bound));
END_ENTITY;
```

Attribute definitions:

— lower_bound optionally specifies a lower bound for valid quantity values for an **nrf_integer_quantity_type**.

— lower_bound_inclusive specifies whether the lower_bound itself is included or excluded from the valid value range.

— upper_bound optionally specifies an upper bound for valid quantity values for an **nrf_integer_quantity_type**.

— upper_bound_inclusive specifies whether the upper_bound itself is included or excluded from the valid value range.

— number_of_real_values is derived to be the number of real quantity values to be held for an **nrf_integer_quantity_type** in the real_values list of nrf_datacube or nrf_any_quantity_value_prescription instances depending on the actual uncertainty_method. It may be 0, 1 or 2.

— number_of_integer_values is derived to be the number of integer quantity values to be held for an **nrf_integer_quantity_type** in the integer_values list of nrf_datacube or

nrf_any_quantity_value_prescription instances depending on the actual uncertainty_method. It is always one.

Formal propositions:

— has_valid_bounds: If both lower_bound and upper_bound are specified then the lower_bound shall be less than or equal to the upper_bound.

4.2.4.32 ENTITY nrf_string_quantity_type

An **nrf_string_quantity_type** is an nrf_any_scalar_quantity_type that specifies a non-physical quantity type whose value is an index that denotes a position in a defined list of string values. The index is an integer value in the range from one to the total number of defined string values.

EXAMPLE An example of an **nrf_string_quantity_type** is 'location_area' with values: 1 for 'Amsterdam', 2 for 'Paris', 3 for 'London', 4 for 'Europe', 5 for 'Antarctica'.

Express specification:

```
ENTITY nrf_string_quantity_type
  SUBTYPE OF(nrf_any_scalar_quantity_type);
  string_values : LIST OF STRING;
DERIVE
  SELF\nrf_any_quantity_type.number_of_real_values : INTEGER := 0;
  SELF\nrf_any_quantity_type.number_of_integer_values : INTEGER := 1;
END_ENTITY;
```

Attribute definitions:

— string_values specifies the list of zero or more string values defined for an **nrf_string_quantity_type**.

— number_of_real_values specifies the total number of real quantity values to be held in an **nrf_string_quantity_type**, which is always zero.

— number_of_integer_values specifies the total number of integer quantity values to be held in an **nrf_string_quantity_type**, which is always one.

4.2.4.33 ENTITY nrf_enumeration_quantity_type

An **nrf_enumeration_quantity_type** is an nrf_any_scalar_quantity_type that specifies a non-physical quantity type whose value is an index that denotes a position in a defined list of enumeration string values and descriptions. The index is an integer value in the range from one to the total number of defined enumeration_items.

EXAMPLE An **nrf_enumeration_quantity_type** may be used to represent the quantity 'operational_mode' of a system with enumeration values: 'switched-off', 'hibernating', 'stand-by', 'operational'.

Express specification:

```
ENTITY nrf_enumeration_quantity_type
  SUBTYPE OF(nrf_any_scalar_quantity_type);
  enumeration_items : LIST OF nrf_enumeration_item;
DERIVE
```



```
  SELF\nrf_any_quantity_type.number_of_real_values : INTEGER := 0;
  SELF\nrf_any_quantity_type.number_of_integer_values : INTEGER := 1;
WHERE
  has_unique_enumeration_names: nrf_verify_unique_names_in_enumeration_item_list(SELF);
END_ENTITY;
```

Attribute definitions:

- enumeration_items specifies the list of valid enumeration string values with description, for an **nrf_enumeration_quantity_type**.
- number_of_real_values specifies the total number of real quantity values to be held in an **nrf_enumeration_quantity_type**, which is always zero.
- number_of_integer_values specifies the total number of integer quantity values to be held in an **nrf_enumeration_quantity_type**, which is always one.

Formal propositions:

- has_unique_enumeration_names: The names of the enumeration_items shall be unique.

4.2.4.34 ENTITY nrf_enumeration_item

An **nrf_enumeration_item** specifies an enumeration item entry for the enumeration_items list of an nrf_enumeration_quantity_type. It consists of a (typically) short name and a more extensive textual description.

Express specification:

```
ENTITY nrf_enumeration_item;
  name        : nrf_label;
  description : nrf_text;
END_ENTITY;
```

Attribute definitions:

- name specifies the name (i.e. the string value) of the enumeration item.
- description specifies the textual description of the enumeration item. This may be the explanation or definition of its usage or it may be the empty string if the name is self-explanatory.

4.2.4.35 FUNCTION nrf_verify_unique_names_in_enumeration_item_list

The function **nrf_verify_unique_names_in_enumeration_item_list** verifies that the nrf_enumeration_item instances in the enumeration_items of an nrf_enumeration_quantity_type all have a unique name. The function returns TRUE if this is the case, and FALSE otherwise.

Express specification:

```
FUNCTION nrf_verify_unique_names_in_enumeration_item_list(
    eqt : nrf_enumeration_quantity_type) : BOOLEAN;
  LOCAL
    name_list : LIST OF nrf_label := [];
  END_LOCAL;
  REPEAT i:=1 TO SIZEOF(eqt.enumeration_items);
    name_list := name_list + eqt.enumeration_items[i].name;
```



```
  END_REPEAT;
  IF VALUE_UNIQUE(name_list) THEN
    RETURN(TRUE);
  ELSE
    RETURN(FALSE);
  END_IF;
END_FUNCTION;
```

Argument definitions:

— eil specifies the candidate list of nrf_enumeration_item to be verified.

4.2.4.36 ENTITY nrf_tensor_characteristic

An nrf_tensor_characteristics specifies a characteristic of a tensor quantity type.

EXAMPLE Examples of names of tensor characteristics are: 'vector', 'matrix', 'covariant', 'contravariant', 'mixed', 'affine', 'complete symmetric', 'Jacobian', 'cartesian'.

Express specification:

```
ENTITY nrf_tensor_characteristic;
  name        : nrf_non_blank_label;
  description : nrf_text;
UNIQUE
  has_unique_name: name;
END_ENTITY;
```

Formal propositions:

— has_unique_name: The name shall be unique in the dataset.

4.2.4.37 ENTITY nrf_tensor_element

An **nrf_tensor_element** specifies the name and quantity type of an element of an nrf_any_tensor_quantity_type.

Express specification:

```
ENTITY nrf_tensor_element;
  name          : nrf_non_blank_label;
  quantity_type : nrf_any_scalar_quantity_type;
END_ENTITY;
```

Attribute definitions:

— name specifies the name for the tensor element.

— quantity_type specifies the scalar quantity type for the tensor element.

4.2.4.38 ENTITY nrf_any_tensor_quantity_type

An **nrf_any_tensor_quantity_type** is an abstract supertype that provides a generic mechanism to reference an nrf_general_tensor_quantity_type, an nrf_symmetric_matrix_quantity_type or an nrf_anti_symmetric_matrix_quantity_type. A tensor quantity type specifies a compound quantity type

containing multiple scalar quantity elements, possibly in more than one dimension. A tensor has a rank which defines the number of dimensions. A tensor of rank one is usually called a vector and a tensor of rank two is usually called a matrix. Tensors of rank three or higher are called higher order tensors. The tensor's elements are accessed using an index. The index tuple has as many elements as the tensor has dimensions. The tensor's quantity values are stored in row-major order in the real_values and/or integer_values list attributes of nrf_datacube, nrf_tensor_quantity_value_prescription or nrf_state_list instances. The storage order of quantity values of a tensor in the real_values and/or integer_values of an nrf_datacube or nrf_any_quantity_value_prescription) is a flat concatenation of element values with the index of the first dimension varying slowest and the index of last dimension varying fastest, i.e. row-major order, the same as the storage convention for multi-dimensional arrays in the C programming language.

NOTE In this protocol the concept of tensor is taken beyond its strict mathematical definition in the sense that its elements may be non-physical quantities: enumeration or string quantities as described earlier. The **nrf_any_tensor_quantity_type** enables the definition in a STEP-NRF dictionary of compound quantity types similar to record datatypes with named fields as commonly used in data processing systems.

Express specification:

```
ENTITY nrf_any_tensor_quantity_type
  ABSTRACT SUPERTYPE OF(ONEOF(
    nrf_general_tensor_quantity_type,
    nrf_symmetric_matrix_quantity_type,
    nrf_anti_symmetric_matrix_quantity_type))
  SUBTYPE OF(nrf_any_quantity_type);
  characteristics : LIST [0:?] OF UNIQUE nrf_tensor_characteristic;
  dimensions      : LIST [1:?] OF nrf_positive_integer;
  elements        : LIST [1:?] OF nrf_tensor_element;
DERIVE
  SELF\nrf_any_quantity_type.rank : INTEGER := SIZEOF(dimensions);
  SELF\nrf_any_quantity_type.number_of_real_values : INTEGER :=
    nrf_get_number_of_real_values_in_any_tensor(SELF);
  SELF\nrf_any_quantity_type.number_of_integer_values : INTEGER :=
    nrf_get_number_of_integer_values_in_any_tensor(SELF);
WHERE
  has_valid_number_of_elements: (SIZEOF(elements) = 1) OR
    (SIZEOF(elements) = nrf_get_required_number_of_elements_in_any_tensor(SELF));
END_ENTITY;
```

Attribute definitions:

- characteristics specifies zero or more textually defined characteristics of an nrf_tensor_quantity_category.
- dimensions specifies both the number of dimensions of the tensor quantity type as well as the number of elements for each dimension.
- elements specifies the scalar quantity type elements that the tensor quantity type comprises. It is allowed to specify only one element: in this special case all elements of the tensor are defined to have the same scalar quantity type, otherwise an explicit list of nrf_tensor_element must be given for all elements of the tensor. Through the definitions nrf_tensor_element instances optionally a name can be assigned to each element.
- rank is derived to be equal to the number of dimensions of the tensor quantity type.
- number_of_real_values is derived to be the total number of concatenated real values required by all quantity types specified in elements.
- number_of_integer_values is derived to be the total number of concatenated integer values required by all quantity types specified in elements.

Formal propositions:

— has_valid_number_of_elements: The number of elements shall be one or the required number of elements for the specific tensor quantity type.

4.2.4.39 ENTITY nrf_general_tensor_quantity_type

An **nrf_general_tensor_quantity_type** is a type of nrf_any_tensor_quantity_type that specifies a quantity type that is general tensor of any rank. The storage order of quantity values for an **nrf_general_tensor_quantity_type** (in the real_values and/or integer_values of an nrf_datacube or nrf_any_quantity_value_prescription) is a flat concatenation of element values with the index of the first dimension varying slowest and the index of last dimension varying fastest, i.e. row-major order, the same as the storage convention for multi-dimensional arrays in the C programming language.

Express specification:

```
ENTITY nrf_general_tensor_quantity_type
  SUBTYPE OF(nrf_any_tensor_quantity_type);
END_ENTITY;
```

4.2.4.40 ENTITY nrf_symmetric_matrix_quantity_type

An **nrf_symmetric_matrix_quantity_type** is a type of nrf_any_tensor_quantity_type that specifies a quantity type that is a symmetric square matrix, i.e. a tensor of rank 2. For the elements of a symmetric square matrix **A** the following relation holds: $a_{ij} = a_{ji}$. In order to reduce storage space only the elements and values on the diagonal and in the upper triangle of the matrix will be defined. An N×N symmetric matrix holds N(N+1)/2 non-redundant elements. The storage order of the quantity values for an **nrf_symmetric_matrix_quantity_type** (in the real_values and/or integer_values of an nrf_datacube, nrf_tensor_quantity_value_prescription or nrf_state_list) is the upper triangle of the matrix, i.e. a flat concatenation of the first full row, followed by the element values from the diagonal element to the last column of all subsequent rows, therefore ending with the last diagonal element value.

Express specification:

```
ENTITY nrf_symmetric_matrix_quantity_type
  SUBTYPE OF(nrf_any_tensor_quantity_type);
WHERE
  is_rank_two: SELF\nrf_any_tensor_quantity_type.rank = 2;
  is_square_matrix: dimensions[1] = dimensions[2];
END_ENTITY;
```

Formal propositions:

— is_rank_two: The rank shall be two.

— is_square_matrix: The first and second dimension shall be equal, i.e. it shall be a square matrix.

4.2.4.41 ENTITY nrf_anti_symmetric_matrix_quantity_type

An **nrf_anti_symmetric_matrix_quantity_type** is a type of nrf_any_tensor_quantity_type that specifies a quantity type that is an anti-symmetric square matrix, i.e. a tensor of rank 2. For the elements of an anti-symmetric square matrix **A** the following relation holds: $a_{ij} = -a_{ji}$. In order to reduce storage space only the elements and values on the diagonal and in the upper triangle of the matrix will be defined. An N×N anti-

symmetric matrix holds N(N+1)/2 non-redundant elements. The storage order of real or integer values for an nrf_symmetric_matrix_quantity_type (in the real_values and/or integer_values of an nrf_datacube or nrf_any_quantity_value_prescription) is a flat concatenation of the first full row, followed by the element values from the diagonal element to the last column of all subsequent rows, therefore ending with the last diagonal element value.

Express specification:

```
ENTITY nrf_anti_symmetric_matrix_quantity_type
  SUBTYPE OF(nrf_any_tensor_quantity_type);
WHERE
  is_rank_two: SELF\nrf_any_tensor_quantity_type.rank = 2;
  is_square_matrix: dimensions[1] = dimensions[2];
END_ENTITY;
```

Formal propositions:

- is_rank_two: The rank shall be two.
- is_square_matrix: The first and second dimension shall be equal, i.e. it shall be a square matrix.

4.2.4.42 FUNCTION nrf_get_qualified_quantity_name

The function **nrf_get_qualified_quantity_name** derives the fully qualified name for an nrf_qualified_physical_quantity_category. The fully qualified name is the concatenation of the names of the qualifiers and the name of the basic_category, joined with underscore characters.

Express specification:

```
FUNCTION nrf_get_qualified_quantity_name(
    qpqc : nrf_qualified_physical_quantity_category) : nrf_non_blank_label;
  LOCAL
    qualified_name : STRING := '';
  END_LOCAL;
  REPEAT i := 1 TO SIZEOF(qpqc.qualifiers);
    IF i = 1 THEN
      qualified_name := qpqc.qualifiers[i].name;
    ELSE
      qualified_name := qualified_name + '_' + qpqc.qualifiers[i].name;
    END_IF;
  END_REPEAT;
  qualified_name := qualified_name + '_' + qpqc.basic_category.name;
  RETURN(qualified_name);
END_FUNCTION;
```

Argument definitions:

- qpqc specifies the nrf_qualified_physical_quantity_category for which the fully qualified name shall be returned.

4.2.4.43 FUNCTION nrf_get_qualified_quantity_symbol

The function **nrf_get_qualified_quantity_symbol** derives the fully qualified symbol of an nrf_qualified_physical_quantity_category. The fully qualified symbol is the concatenation of the symbol of the quantity_category and the symbols of the qualifiers in reverse order, joined with underscore characters.

Express specification:

```
FUNCTION nrf_get_qualified_quantity_symbol(
    qpqc : nrf_qualified_physical_quantity_category) : nrf_non_blank_label;
  LOCAL
    qualified_symbol : nrf_non_blank_label;
  END_LOCAL;
  qualified_symbol := qpqc.basic_category.symbol;
  REPEAT i := SIZEOF(qpqc.qualifiers) TO 1 BY -1;
    qualified_symbol := qualified_symbol + '_' + qpqc.qualifiers[i].symbol;
  END_REPEAT;
  RETURN(qualified_symbol);
END_FUNCTION;
```

Argument definitions:

— qt specifies the nrf_any_quantity_type for which the fully qualified symbol shall be returned.

4.2.4.44 FUNCTION nrf_get_qualified_quantity_description

The function **nrf_get_qualified_quantity_description** derives the fully qualified description for an nrf_qualified_physical_quantity_category. The fully qualified description consists of the description of the basic_category, followed by the text " -- qualified by:" and then the names and descriptions of each of the qualifiers separated by semicolons.

Express specification:

```
FUNCTION nrf_get_qualified_quantity_description(
    qpqc : nrf_qualified_physical_quantity_category) : nrf_text;
  LOCAL
    qualified_description : STRING := '';
  END_LOCAL;
  qualified_description := qpqc.basic_category.description + ' -- qualified by:';
  REPEAT i := 1 TO SIZEOF(qpqc.qualifiers);
    qualified_description := qualified_description
      + ' ' + qpqc.qualifiers[i].name
      + ' - ' + qpqc.qualifiers[i].description + ';';
  END_REPEAT;
  RETURN(qualified_description);
END_FUNCTION;
```

Argument definitions:

— qpqc specifies the nrf_qualified_physical_quantity_category for which the fully qualified description shall be returned.

4.2.4.45 FUNCTION nrf_get_required_number_of_elements_in_any_tensor

The function **nrf_get_required_number_of_elements_in_any_tensor** yields the required number of elements for an nrf_any_tensor_quantity_type.

Express specification:

```
FUNCTION nrf_get_required_number_of_elements_in_any_tensor(
    atqt : nrf_any_tensor_quantity_type): INTEGER;
  LOCAL
    number_of_elements : INTEGER := 1;
```



```
  END_LOCAL;
  IF 'NRF_ARM.NRF_GENERAL_TENSOR_QUANTITY_TYPE' IN TYPEOF(atqt) THEN
    IF SIZEOF(atqt.elements) = 1 THEN
      REPEAT i := 1 TO SIZEOF(atqt.dimensions);
        number_of_elements := number_of_elements * atqt.dimensions[i];
      END_REPEAT;
    ELSE
      number_of_elements := SIZEOF(atqt.elements);
    END_IF;
  ELSE
    -- atqt is an nrf_symmetric_matrix_quantity_type
    -- or an nrf_anti_symmetric_matrix_quantity_type
    number_of_elements := atqt.dimensions[1] * (atqt.dimensions[1] + 1) DIV 2;
  END_IF;
  RETURN(number_of_elements);
END_FUNCTION;
```

Argument definitions:

— atqt specifies the nrf_any_tensor_quantity_type for which to yield the required number of elements.

4.2.4.46 FUNCTION nrf_get_number_of_real_values_for_real_quantity_type

The function **nrf_get_number_of_real_values_for_real_quantity_type** computes and returns the number of real values required for a given nrf_real_quantity_type.

Express specification:

```
FUNCTION nrf_get_number_of_real_values_for_real_quantity_type(
    qt : nrf_real_quantity_type): INTEGER;
  LOCAL
    number_of_real_values : INTEGER;
  END_LOCAL;
  IF EXISTS(qt.uncertainty_method) THEN
    CASE qt.uncertainty_method.margins OF
      nrf_uncertainty_margins_type.SYMMETRIC_ABSOLUTE:
        number_of_real_values := 2;
      nrf_uncertainty_margins_type.SYMMETRIC_PERCENTAGE:
        number_of_real_values := 2;
      nrf_uncertainty_margins_type.ASYMMETRIC_ABSOLUTE:
        number_of_real_values := 3;
      nrf_uncertainty_margins_type.ASYMMETRIC_PERCENTAGE:
        number_of_real_values := 3;
    END_CASE;
  ELSE
    number_of_real_values := 1;
  END_IF;
  RETURN(number_of_real_values);
END_FUNCTION;
```

Argument definitions:

— qt specifies the nrf_real_quantity_type for which the number of real values must be computed.

4.2.4.47 FUNCTION nrf_get_number_of_real_values_for_integer_quantity_type

The function **nrf_get_number_of_real_values_for_integer_quantity_type** computes and returns the number of real values required for a given nrf_integer_quantity_type.

Express specification:

```
FUNCTION nrf_get_number_of_real_values_for_integer_quantity_type(
    qt : nrf_integer_quantity_type): INTEGER;
  LOCAL
    number_of_real_values : INTEGER;
  END_LOCAL;
  IF EXISTS(qt.uncertainty_method) THEN
    CASE qt.uncertainty_method.margins OF
      nrf_uncertainty_margins_type.SYMMETRIC_ABSOLUTE:
        number_of_real_values := 1;
      nrf_uncertainty_margins_type.SYMMETRIC_PERCENTAGE:
        number_of_real_values := 1;
      nrf_uncertainty_margins_type.ASYMMETRIC_ABSOLUTE:
        number_of_real_values := 2;
      nrf_uncertainty_margins_type.ASYMMETRIC_PERCENTAGE:
        number_of_real_values := 2;
    END_CASE;
  ELSE
    number_of_real_values := 0;
  END_IF;
  RETURN(number_of_real_values);
END_FUNCTION;
```

Argument definitions:

— qt specifies the nrf_integer_quantity_type for which the number of real values must be computed.

4.2.4.48 FUNCTION nrf_get_number_of_real_values_in_any_tensor

The function **nrf_get_number_of_real_values_in_any_tensor** computes and returns the total number of real values for the elements that have been specified for an nrf_any_tensor_quantity_type.

Express specification:

```
FUNCTION nrf_get_number_of_real_values_in_any_tensor(
    atqt : nrf_any_tensor_quantity_type): INTEGER;
  LOCAL
    total_number_of_real_values : INTEGER := 0;
  END_LOCAL;
  IF SIZEOF(atqt.elements) = 1 THEN
    total_number_of_real_values :=
      nrf_get_required_number_of_elements_in_any_tensor(atqt) *
      atqt.elements[1].quantity_type.number_of_real_values;
  ELSE
    REPEAT i := 1 TO SIZEOF(atqt.elements);
      total_number_of_real_values := total_number_of_real_values +
        atqt.elements[i].quantity_type.number_of_real_values;
    END_REPEAT;
  END_IF;
  RETURN(total_number_of_real_values);
END_FUNCTION;
```

Argument definitions:

— atqt specifies the nrf_any_tensor_quantity_type for which the number of real values must be computed.

4.2.4.49 FUNCTION nrf_get_number_of_integer_values_in_any_tensor

The function **nrf_get_number_of_integer_values_in_any_tensor** computes and returns the total number of integer values for the elements that have been specified for an nrf_any_tensor_quantity_type.

Express specification:

```
FUNCTION nrf_get_number_of_integer_values_in_any_tensor(
    atqt : nrf_any_tensor_quantity_type): INTEGER;
  LOCAL
    total_number_of_integer_values : INTEGER := 0;
  END_LOCAL;
  IF SIZEOF(atqt.elements) = 1 THEN
    total_number_of_integer_values :=
      nrf_get_required_number_of_elements_in_any_tensor(atqt) *
      atqt.elements[1].quantity_type.number_of_integer_values;
  ELSE
    REPEAT i := 1 TO SIZEOF(atqt.elements);
      total_number_of_integer_values := total_number_of_integer_values +
        atqt.elements[i].quantity_type.number_of_integer_values;
    END_REPEAT;
  END_IF;
  RETURN(total_number_of_integer_values);
END_FUNCTION;
```

Argument definitions:

— atqt specifies the nrf_any_tensor_quantity_type for which the number of integer values must be computed.

4.2.4.50 ENTITY nrf_quantity_type_list

An **nrf_quantity_type_list** specifies an ordered list of references to nrf_any_quantity_type instances. It can be used as the quantity_type_basis of an nrf_datacube.

Express specification:

```
ENTITY nrf_quantity_type_list;
  id             : nrf_identifier;
  name           : nrf_label;
  description    : nrf_text;
  quantity_types : LIST [1:?] OF UNIQUE nrf_any_quantity_type;
DERIVE
  number_of_real_values : INTEGER :=
    nrf_derive_number_of_real_values_in_quantity_type_list(SELF);
  number_of_integer_values : INTEGER :=
    nrf_derive_number_of_integer_values_in_quantity_type_list(SELF);
END_ENTITY;
```

Attribute definitions:

— id specifies the identifier of an instance of **nrf_quantity_type_list**.
— name specifies the human-interpretable name of an instance of **nrf_quantity_type_list**.
— description specifies the textual description of an instance of **nrf_quantity_type_list**.
— quantity_types specifies the list of quantity types.
— number_of_values specifies the total number of required values for the quantity_types.

4.2.4.51 FUNCTION nrf_derive_number_of_real_values_in_quantity_type_list

The function **nrf_derive_number_of_real_values_in_quantity_type_list** computes and returns the total number of required real values for the list of quantity types defined in an nrf_quantity_type_list.

Express specification:

```
FUNCTION nrf_derive_number_of_real_values_in_quantity_type_list(
    qtl : nrf_quantity_type_list) : INTEGER;
  LOCAL
    n : INTEGER := 0;
  END_LOCAL;
  REPEAT i := 1 TO SIZEOF(qtl.quantity_types);
    n := n + qtl.quantity_types[i].number_of_real_values;
  END_REPEAT;
  RETURN(n);
END_FUNCTION;
```

Argument definitions:

— qtl specifies the nrf_quantity_type_list for which the total number of real values has to be computed.

4.2.4.52 FUNCTION nrf_derive_number_of_integer_values_in_quantity_type_list

The function **nrf_derive_number_of_integer_values_in_quantity_type_list** computes and returns the total number of required integer values for the list of quantity types defined in an nrf_quantity_type_list.

Express specification:

```
FUNCTION nrf_derive_number_of_integer_values_in_quantity_type_list(
    qtl : nrf_quantity_type_list) : INTEGER;
  LOCAL
    n : INTEGER := 0;
  END_LOCAL;
  REPEAT i := 1 TO SIZEOF(qtl.quantity_types);
    n := n + qtl.quantity_types[i].number_of_integer_values;
  END_REPEAT;
  RETURN(n);
END_FUNCTION;
```

Argument definitions:

— qtl specifies the nrf_quantity_type_list for which the total number of integer values has to be computed.

4.2.5 NRF Date and time UoF

The date_and_time UoF specifies various representations for calendar date and time of day.

NOTE The objects defined in this UoF are defined to map one-to-one with those in the date_time_schema of ISO 10303-41. The ISO 10303-41 date_time_schema objects ordinal_date, day_of_week and week_of_year are out of scope.

4.2.5.1 TYPE nrf_ahead_or_behind

The ahead_or_behind type is used to specify whether a given time is ahead of or behind coordinated universal time.

NOTE Coordinated Universal Time (UTC) is the international time standard. It is the current term for what was commonly referred to as Greenwich Meridian Time (GMT). Zero (0) hours UTC is midnight in Greenwich England, which lies on the zero longitudinal meridian. Coordinated universal time is based on a 24 hour clock; therefore, afternoon hours such as 4 PM are expressed as 16:00 UTC (sixteen hours, zero minutes).

Express specification:

```
TYPE nrf_ahead_or_behind = ENUMERATION OF(
  AHEAD,
  EXACT,
  BEHIND);
END_TYPE;
```

4.2.5.2 TYPE nrf_year_number

A year_number is the year as defined in the Gregorian calendar. The year_number shall be completely and explicitly specified using as many digits as necessary to unambiguously convey the century and year within the century. Truncated year numbers shall not be used.

NOTE ISO 8601:1988 defines the Gregorian calendar.

EXAMPLE The year_number corresponding to the first manned landing on the moon is 1969 (not 69).

Express specification:

```
TYPE nrf_year_number = INTEGER;
END_TYPE;
```

4.2.5.3 TYPE nrf_month_in_year_number

An **nrf_month_in_year_number** specifies the position of the specified month in a year as defined in ISO 8601 (subclause 5.2.1).

NOTE January is month number 1, February is month number 2, March is month number 3, April is month number 4, May is month number 5, June is month number 6, July is month number 7, August is month number 8, September is month number 9, October is month number 10, November is month number 11, and December is month number 12.

Express specification:

```
TYPE nrf_month_in_year_number = INTEGER;
WHERE
  is_valid_month_number: { 1 <= SELF <= 12 };
END_TYPE;
```

Formal propositions:

— is_valid_month_number: The value of the integer shall be between 1 and 12.

4.2.5.4 TYPE nrf_day_in_month_number

An **nrf_day_in_month_number** specifies the position of the specified day in a month.

Express specification:

```
TYPE nrf_day_in_month_number = INTEGER;
WHERE
  is_valid_day_in_month_number: {1 <= SELF <= 31};
END_TYPE;
```

Formal propositions:

— is_valid_day_in_month_number: The value of the integer shall be between 1 and 31.

4.2.5.5 TYPE nrf_hour_in_day

An **nrf_hour_in_day** specifies the hour element of a specified time on a 24 hour clock. Midnight shall be represented by the value zero.

EXAMPLE The hour_in_day corresponding to 3 o'clock in the afternoon is 15.

NOTE Although ISO 8601 allows two representations for midnight, 0000 and 2400, this part of ISO 10303 restricts the representation to the first value.

Express specification:

```
TYPE nrf_hour_in_day = INTEGER;
WHERE
  is_valid_hour_in_day: {0 <= SELF < 24};
END_TYPE;
```

Formal propositions:

— is_valid_hour_in_day: The value of the integer shall be between 0 and 23.

4.2.5.6 TYPE nrf_minute_in_hour

An **nrf_minute_in_hour** specifies the minute element of a specified time.

Express specification:

```
TYPE nrf_minute_in_hour = INTEGER;
WHERE
  is_valid_minute_in_hour: {0 <= SELF < 60};
END_TYPE;
```

Formal propositions:

— is_valid_minute_in_hour: The value of the integer shall be between 0 and 59.

4.2.5.7 TYPE nrf_second_in_minute

An **nrf_second_in_minute** specifies the second element of a specified time.

Express specification:

```
TYPE nrf_second_in_minute = REAL;
WHERE
  is_valid_second_in_minute: {0.0 <= SELF <= 60.0};
END_TYPE;
```

Formal propositions:

— is_valid_second_in_minute: The value of the real number shall be between 0.0 to 60.0.

NOTE A value of 60 allows for leap seconds.

NOTE The mean solar time is determined by the rotation of the earth. Leap seconds are added as required, usually in the middle or at the end of a year, and ensure that the legal time does not differ from the nonuniform mean solar time by more than one second, in spite of the variations of the earth rotation.

4.2.5.8 ENTITY nrf_calendar_date

An **nrf_calendar_date** specifies a type of date defined as a day in a month of a year.

Express specification:

```
ENTITY nrf_calendar_date;
  year_component  : nrf_year_number;
  month_component : nrf_month_in_year_number;
  day_component   : nrf_day_in_month_number;
WHERE
  is_valid_calendar_date: nrf_verify_calendar_date(SELF);
END_ENTITY;
```

Attribute definitions:

— year_component specifies the year in which the date occurs.

— month_component specifies the month element of the date.

— day_component specifies the day element of the date.

Formal propositions:

— is_valid_calendar_date: The entity shall define a valid **nrf_calendar_date**.

4.2.5.9 ENTITY nrf_coordinated_universal_time_offset

A **nrf_coordinated_universal_time_offset** specifies the oriented offset (specified in hours and possibly minutes) from the coordinated universal time. The offset value shall be positive.

NOTE Coordinated Universal Time (UTC) is the international time standard. It is the current term for what was commonly referred to as Greenwich Meridian Time (GMT). Zero (0) hours UTC is midnight in Greenwich England, which lies on the zero longitudinal meridian. Coordinated universal time is based on a

24 hour clock; therefore, afternoon hours such as 4 pm are expressed as 16:00 UTC (sixteen hours, zero minutes).

NOTE An **nrf_coordinated_universal_time_offset** is used to relate a time to coordinated universal time by an offset (specified in hours and minutes) and a direction.

Express specification:

```
ENTITY nrf_coordinated_universal_time_offset;
  hour_offset   : nrf_hour_in_day;
  minute_offset : OPTIONAL nrf_minute_in_hour;
  sense         : nrf_ahead_or_behind;
DERIVE
  actual_minute_offset: INTEGER := NVL(minute_offset, 0);
WHERE
  has_valid_actual_minute_offset: { 0 <= actual_minute_offset <= 59 };
  has_valid_offsets: NOT (
    ((hour_offset <> 0) OR (actual_minute_offset <>0))
    AND (sense = exact));
END_ENTITY;
```

Attribute definitions:

- hour_offset specifies the number of hours by which a time is offset from coordinated universal time.
- minute_offset optionally specifies the number of minutes by which a time is offset from coordinated universal time.
- sense specifies the direction of the offset.
- actual_minute_offset specifies the value of the number of minutes offset used to compute the **nrf_coordinated_universal_time_offset**, either the value of nrf_minute_offset or 0.

Formal propositions:

- has_valid_actual_minute_offset: The actual_minute_offset shall be a positive number, less than or equal to 59.
- has_valid_offsets: If the value of sense specifies that there is no offset from the Coordinated Universal time, hour_offset and actual_minute_offset shall both be equal to zero. If either hour_offset or actual_minute_offset is different from zero, the value of sense shall specify that there is an offset, either ahead or behind, from the Coordinated Universal time.

4.2.5.10 ENTITY nrf_local_time

An **nrf_local_time** specifies a moment in time on a 24-hour clock by hour, minute, and second. The instance is expressed in the local time zone and the offset with the coordinate universal time shall be specified.

NOTE This construct is used to represent a moment in (wall clock) time whereas an nrf_real_quantity_type for a quantity_category 'time' is used to represent an amount of time (on some relative time scale).

EXAMPLE 15:00 hours is a moment in time whereas 15 hours is an amount of time.

Express specification:

```
ENTITY nrf_local_time;
  hour_component   : nrf_hour_in_day;
  minute_component : OPTIONAL nrf_minute_in_hour;
```



```
  second_component : OPTIONAL nrf_second_in_minute;
  zone             : nrf_coordinated_universal_time_offset;
WHERE
  is_valid_local_time: nrf_verify_time(SELF);
END_ENTITY;
```

Attribute definitions:

- hour_component specifies the number of hours
- minute_component optionally specifies the number of minutes
- second_component optionally specifies the number of seconds
- zone specifies the offset of the local time zone to the coordinated universal time.

Formal propositions:

- is_valid_local_time: The entity shall define a valid time.

4.2.5.11 ENTITY nrf_date_and_time

A date_and_time specifies a moment in time on a particular day.

Express specification:

```
ENTITY nrf_date_and_time;
  date_component : nrf_calendar_date;
  time_component : nrf_local_time;
END_ENTITY;
```

Attribute definitions:

- date_component specifies the date element of the date time combination.
- time_component specifies the time element of the date time combination.

4.2.5.12 FUNCTION nrf_verify_calendar_date

The **nrf_verify_calendar_date** function determines whether the components of an nrf_calendar_date indicate a valid date. If the nrf_calendar_date is valid, the function returns TRUE. Otherwise it returns FALSE.

Express specification:

```
FUNCTION nrf_verify_calendar_date(date : nrf_calendar_date) : LOGICAL;
  CASE date.month_component OF
    1 : RETURN({ 1 <= date.day_component <= 31 });
    2 : BEGIN
          IF (nrf_verify_leap_year(date.year_component)) THEN
            RETURN({ 1 <= date.day_component <= 29 });
          ELSE
            RETURN({ 1 <= date.day_component <= 28 });
          END_IF;
        END;
    3 : RETURN({ 1 <= date.day_component <= 31 });
    4 : RETURN({ 1 <= date.day_component <= 30 });
    5 : RETURN({ 1 <= date.day_component <= 31 });
```

```
    6 : RETURN({ 1 <= date.day_component <= 30 });
    7 : RETURN({ 1 <= date.day_component <= 31 });
    8 : RETURN({ 1 <= date.day_component <= 31 });
    9 : RETURN({ 1 <= date.day_component <= 30 });
    10: RETURN({ 1 <= date.day_component <= 31 });
    11: RETURN({ 1 <= date.day_component <= 30 });
    12: RETURN({ 1 <= date.day_component <= 31 });
  END_CASE;
  RETURN(FALSE);
END_FUNCTION; -- nrf_verify_calendar_date
```

Argument definitions:

— date specifies the candidate nrf_calendar_date that is to be verified.

4.2.5.13 FUNCTION nrf_verify_time

The **nrf_verify_time** function determines whether a candidate nrf_local_time has a minute_component if it has a second_component. It returns FALSE if the condition is not met. Otherwise it returns TRUE.

Express specification:

```
FUNCTION nrf_verify_time(a_time: nrf_local_time) : BOOLEAN;
  IF EXISTS(a_time.second_component) THEN
    RETURN(EXISTS(a_time.minute_component));
  ELSE
    RETURN(TRUE);
  END_IF;
END_FUNCTION;
```

Argument definitions:

— a_time specifies the candidate nrf_local_time that is to be verified.

4.2.5.14 FUNCTION nrf_verify_leap_year

The leap_year function determines whether a given year is a leap year or not according to the Gregorian calendar algorithm. It returns TRUE if the year is a leap year. Otherwise, it returns FALSE.

Express specification:

```
FUNCTION nrf_verify_leap_year(year : nrf_year_number) : BOOLEAN;
  IF ((((year MOD 4) = 0) AND ((year MOD 100) <> 0)) OR ((year MOD 400) = 0)) THEN
    RETURN(TRUE);
  ELSE
    RETURN(FALSE);
  END_IF;
END_FUNCTION;
```

Argument definitions:

— year specifies the candidate nrf_year_number that is being verified.

4.2.6 NRF Parametrics UoF

The nrf_parametrics UoF collects all application objects that enable the specification of named variables, mathematical expressions and functions.

4.2.6.1 TYPE nrf_algorithmic_expression

An **nrf_algorithmic_expression** specifies a string that holds an algorithmic expression in the syntax of a given nrf_algorithmic_language.

Express specification:

```
TYPE nrf_algorithmic_expression = STRING;
END_TYPE;
```

4.2.6.2 TYPE nrf_algorithmic_statement

An **nrf_algorithmic_statement** specifies a string that holds an algorithmic statement in the syntax of a given nrf_algorithmic_language.

Express specification:

```
TYPE nrf_algorithmic_statement = STRING;
END_TYPE;
```

4.2.6.3 ENTITY nrf_algorithmic_language

An **nrf_algorithmic_language** specifies the name and description of a programming or scripting language that can be used to define the expressions for an nrf_real_quantity_value_expression, an nrf_integer_quantity_value_expression, an nrf_string_quantity_value_expression, an nrf_enumeration_quantity_value_expression or an nrf_tensor_quantity_value_expression, and the statements for an nrf_model_function instances.

Express specification:

```
ENTITY nrf_algorithmic_language;
  name              : nrf_label;
  version           : nrf_label;
  description       : nrf_text;
  specification_uri : OPTIONAL nrf_uniform_resource_identifier;
UNIQUE
  has_unique_name: name;
END_ENTITY;
```

Attribute definitions:

- name specifies the human-interpretable name of an **nrf_algorithmic_language**.
- version specifies the name of the version of an **nrf_algorithmic_language**.
- description specifies the textual description of an **nrf_algorithmic_language**.
- specification_uri optionally specifies a URI that uniquely identifies the formal specification or standard for the algorithmic language.

Formal propositions:

— has_unique_name: The name shall be unique in the dataset.

4.2.6.4 ENTITY nrf_variable

An **nrf_variable** specifies a named variable that can be referenced in expressions or algorithmic statements. Optionally its value may be prescribed. If a value is prescribed then the variable is considered to be a bound or independent variable, otherwise it is considered an unbound or dependent variable. The name space (or scope) of an **nrf_variable** is the nrf_network_model, nrf_case or nrf_model_function to which it is associated.

Express specification:

```
ENTITY nrf_variable
  SUPERTYPE OF(ONEOF(nrf_formal_parameter));
  id            : nrf_identifier;
  description   : nrf_text;
  quantity_type : nrf_any_quantity_type;
  prescription  : OPTIONAL nrf_any_quantity_value_prescription;
WHERE
  has_valid_quantity_type:
    (NOT EXISTS(prescription)) XOR (quantity_type :=: prescription.quantity_type);
END_ENTITY;
```

Attribute definitions:

— id specifies the identifier by which the variable is known.

— description specifies a textual description of the intent or usage of the variable.

— quantity_type specifies the quantity type of the variable.

— prescription optionally specifies the prescribed value of the variable.

Formal propositions:

— has_valid_quantity_type: If the prescription exists its quantity_type shall be the same as the quantity_type of the **nrf_variable**.

4.2.6.5 ENTITY nrf_model_constraint

An **nrf_model_constraint** specifies a constraint for an nrf_network_model in the form of a mathematical equation. The equation shall use variable names that have been declared through the variables attribute of the relevant nrf_network_model. These constraint declarations can be used to specify non-causal mathematical models.

Express specification:

```
ENTITY nrf_model_constraint;
  id                   : nrf_identifier;
  description          : nrf_text;
  language             : nrf_algorithmic_language;
  constraint           : nrf_algorithmic_expression;
  independent_variables : LIST OF nrf_variable;
  creator_tool         : OPTIONAL nrf_tool_or_facility;
END_ENTITY;
```


Attribute definitions:

- id specifies the identifier of the constraint.
- description specifies the textual description of the constraint.
- language specifies the nrf_algorithmic_language in which the mathematical equation expressing the constraint is formulated.
- constraint specifies the mathematical equation that formulates the constraint in a syntax that conforms to the given language.
- independent_variables specifies the independent variables in the constraint as appropriate.
- creator_tool optionally specifies the tool that was used in the creation of the constraint.

4.2.6.6 TYPE nrf_formal_parameter_in_out

An **nrf_formal_parameter_in_out** specifies the three possibilities for the usage of an nrf_formal_parameter in an nrf_any_function: INPUT_ONLY, OUTPUT_ONLY or INPUT_OUTPUT.

Express specification:

```
TYPE nrf_formal_parameter_in_out = ENUMERATION OF(
  INPUT_ONLY,
  OUTPUT_ONLY,
  INPUT_OUTPUT);
END_TYPE;
```

4.2.6.7 ENTITY nrf_formal_parameter

An **nrf_formal_parameter** specifies a formal parameter for use in the signature of an nrf_model_function. If its prescription is specified (see nrf_variable), it is the default value of the parameter.

Express specification:

```
ENTITY nrf_formal_parameter
  SUBTYPE OF(nrf_variable);
  in_out : nrf_formal_parameter_in_out;
END_ENTITY;
```

Attribute definitions:

- in_out specifies whether an **nrf_formal_parameter** is an INPUT_ONLY, OUTPUT_ONLY or INPUT_OUTPUT parameter. The value of an **nrf_formal_parameter** can always be changed inside the body of a nrf_model_function, but such a change of value is only reflected back to the point of invocation when the in_out attribute of the parameter is set to OUTPUT_ONLY or INPUT_OUTPUT.

4.2.6.8 ENTITY nrf_model_function

An **nrf_model_function** specifies a function signature and an executable function body. The body is defined by a list of statements conforming to a given algorithmic language. An **nrf_model_function** typically specifies user-defined logic that is part of an nrf_network_model.

Express specification:


```
ENTITY nrf_model_function;
  id                   : nrf_identifier;
  description          : nrf_text;
  formal_parameters    : LIST OF nrf_formal_parameter;
  result_quantity_type : OPTIONAL nrf_any_quantity_type;
  language             : nrf_algorithmic_language;
  statements           : LIST OF nrf_algorithmic_statement;
  creator_tool         : OPTIONAL nrf_tool_or_facility;
END_ENTITY;
```

Attribute definitions:

- id specifies the identifier of the **nrf_model_function**.
- description specifies the textual description of the **nrf_model_function**.
- formal_parameters specifies the list of formal function parameters. The prescription of any nrf_formal_parameter may be used to specify a default value.
- result_quantity_type specifies the quantity type of the result to be returned by the **nrf_model_function**.
- language specifies the nrf_algorithmic_language in which the body of the **nrf_model_function** is expressed.
- statements specifies the list of statements that constitute the body of the function in a syntax that conforms to the given language.
- creator_tool optionally specifies the tool that was used in the creation of the **nrf_model_function**.

4.2.6.9 ENTITY nrf_any_quantity_value_prescription

An **nrf_any_quantity_value_prescription** is an abstract supertype that provides a generic mechanism to reference quantity value prescriptions for each of the defined quantity types.

Express specification:

```
ENTITY nrf_any_quantity_value_prescription
  ABSTRACT SUPERTYPE OF(ONEOF(
    nrf_real_quantity_value_prescription,
    nrf_integer_quantity_value_prescription,
    nrf_string_quantity_value_prescription,
    nrf_enumeration_quantity_value_prescription,
    nrf_tensor_quantity_value_prescription));
  quantity_type  : nrf_any_quantity_type;
  real_values    : LIST OF REAL;
  integer_values : LIST OF INTEGER;
WHERE
  has_correct_number_of_real_values:
    SIZEOF(real_values) = quantity_type.number_of_real_values;
  has_correct_number_of_integer_values:
    SIZEOF(integer_values) = quantity_type.number_of_integer_values;
END_ENTITY;
```

Attribute definitions:

- quantity_type specifies the quantity type of the prescription.
- real_values specifies the literal real values for the quantity_type. For a literal subtype it specifies the final values, but for an expression subtype it specifies the evaluated default values for the expression. The ordering in real_values is the same as for the real_values attribute of nrf_datacube.

— integer_values specifies the literal integer values for the quantity_type. For a literal subtype it specifies the final values, but for an expression subtype it specifies the evaluated default values for the expression. The ordering in integer_values is the same as for the integer_values attribute of nrf_datacube.

Formal propositions:

— has_correct_number_of_real_values: The number of real_values shall be as required for the quantity_type.
— has_correct_number_of_integer_values: The number of integer_values shall be as required for the quantity_type.

4.2.6.10 ENTITY nrf_real_quantity_value_prescription

An **nrf_real_quantity_value_prescription** is a type of nrf_any_quantity_value_prescription for an nrf_real_quantity_type. It is an abstract supertype that provides a generic mechanism to reference an nrf_real_quantity_literal, an nrf_real_quantity_expression, an nrf_real_univariate_power_series_polynomial_expression or an nrf_real_interpolation_table_expression.

Express specification:

```
ENTITY nrf_real_quantity_value_prescription
  ABSTRACT SUPERTYPE OF(ONEOF(
    nrf_real_quantity_value_literal,
    nrf_real_quantity_value_expression,
    nrf_real_univariate_power_series_polynomial_expression,
    nrf_real_interpolation_table_expression))
  SUBTYPE OF(nrf_any_quantity_value_prescription);
  SELF\nrf_any_quantity_value_prescription.quantity_type :
    nrf_real_quantity_type;
DERIVE
  val : REAL := real_values[1];
END_ENTITY;
```

Attribute definitions:

— val is derived to be the first real value for the quantity_type, i.e. the best estimated value if quantity_type has an associated uncertainty_method. For an nrf_real_quantity_value_literal it specifies the final value, for the other subtypes it specifies the evaluated default value for the expression.

4.2.6.11 ENTITY nrf_real_quantity_value_literal

An **nrf_real_quantity_value_literal** is a type of nrf_real_quantity_value_prescription that specifies a literal real value for an nrf_real_quantity_type.

Express specification:

```
ENTITY nrf_real_quantity_value_literal
  SUBTYPE OF(nrf_real_quantity_value_prescription);
END_ENTITY;
```


4.2.6.12 ENTITY nrf_real_quantity_value_expression

An **nrf_real_quantity_value_expression** is a type of nrf_real_quantity_value_prescription that specifies the prescription of a value for an nrf_real_quantity_type through an expression conforming to a given algorithmic language. The expression may reference one or more nrf_variable instances through their id.

EXAMPLE Typical use of an nrf_quantity_value_expression would be the specification of a temperature dependent material property like thermal conductivity for a material.

Express specification:

```
ENTITY nrf_real_quantity_value_expression
  SUBTYPE OF(nrf_real_quantity_value_prescription);
  language     : nrf_algorithmic_language;
  expression   : nrf_algorithmic_expression;
  creator_tool : OPTIONAL nrf_tool_or_facility;
END_ENTITY;
```

Attribute definitions:

- language specifies the nrf_algorithmic_language in which the expression is expressed.
- expression specifies an algorithmic expression in a syntax that conforms to the algorithmic language specified in the language attribute.
- creator_tool optionally specifies the tool that was used in the creation of the expression.

4.2.6.13 ENTITY nrf_real_univariate_power_series_polynomial_expression

An **nrf_real_univariate_power_series_polynomial_expression** is a type of nrf_quantity_value_expression that specifies a prescription by defining a univariate polynomial expression by a power series of a specified degree with real coefficients. Such a polynomial of degree *N* is defined by the following equation:

$$f(x) = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \alpha_3 x^3 + \cdots + \alpha_{N-1} x^{N-1} + \alpha_N x^N$$

where x is the independent parameter and the α_i are the coefficients.

Express specification:

```
ENTITY nrf_real_univariate_power_series_polynomial_expression
  SUBTYPE OF(nrf_real_quantity_value_prescription);
  id                    : nrf_identifier;
  description           : nrf_text;
  independent_parameter : nrf_variable;
  degree                : nrf_positive_integer;
  coefficients          : LIST [1:?] OF REAL;
WHERE
  number_of_coefficients_matches_degree: SIZEOF(coefficients) = degree + 1;
END_ENTITY;
```

Attribute definitions:

- id specifies the identifier by which the expression is known.
- description specifies a textual description of the polynomial expression.
- independent_parameter specifies the independent parameter for the polynomial expression.

— degree specifies the degree N of the polynomial function.
— coefficients specifies the list of N+1 coefficients of the polynomial function.

Formal propositions:

— number_of_coefficients_matches_degree: The number of coefficients shall be one more than the degree.

4.2.6.14 TYPE nrf_interpolation_type

An **nrf_interpolation_type** specifies one of the following interpolation procedures: polynomial, linear/ logarithmic

Express specification:

```
TYPE nrf_interpolation_type = ENUMERATION OF (
  POLYNOMIAL,
  LINEAR_LOGARITHMIC,
  LOGARITHMIC_LINEAR,
  LOGARITHMIC_LOGARITHMIC);
END_TYPE;
```

4.2.6.15 ENTITY nrf_real_interpolation_table_expression

An **nrf_real_interpolation_table_expression** is a type of nrf_quantity_value_expression that specifies a prescription defined by an interpolation table containing the dependent values for all combinations of one or more discrete bases of independent parameters, and an interpolation method definition.

Express specification:

```
ENTITY nrf_real_interpolation_table_expression
  SUPERTYPE OF(ONEOF(nrf_cyclic_real_interpolation_table_expression))
  SUBTYPE OF(nrf_real_quantity_value_prescription);
  independent_parameters : LIST [1:?] OF nrf_variable;
  interpolation_type     : nrf_interpolation_type;
  interpolation_degree   : nrf_positive_integer;
  interpolation_table    : nrf_real_lookup_table;
WHERE
  has_correct_number_of_independent_parameters:
    SIZEOF(independent_parameters) =
    SIZEOF(interpolation_table.independent_quantity_types);
  has_valid_interpolation_degree:
    (interpolation_type = POLYNOMIAL) OR (interpolation_degree = 1);
  interpolation_table_has_correct_dependent_quantity_type:
    SELF\nrf_real_quantity_value_prescription.quantity_type :=:
    interpolation_table.dependent_quantity_type;
  interpolation_table_has_correct_independent_quantity_types:
    nrf_verify_independent_quantity_types(SELF);
END_ENTITY;
```

Attribute definitions:

— independent_parameters specifies the list of actual independent variable references to be used.
— interpolation_type specifies the type of interpolation to be performed.
— interpolation_degree specifies which degree of interpolation shall be performed.

— interpolation_table specifies the nrf_real_lookup_table that contains the values for the interpolation.

Formal propositions:

— has_correct_number_of_independent_parameters: The number of independent_parameters shall be equal to the number of independent_quantity_types in the associated interpolation_table.
— has_valid_interpolation_degree: If the interpolation_type is POLYNOMIAL any interpolation_degree is permitted, if the interpolation_type is not POLYNOMIAL the interpolation_degree shall be one.
— interpolation_table_has_correct_dependent_quantity_type: The resulting quantity type of the expression shall be the same as the quantity type of the dependent_values in the interpolation_table.
— interpolation_table_has_correct_independent_quantity_types: The quantity types of the independent_parameters shall be the same as the independent_quantity_types of the associated interpolation_table.

4.2.6.16 ENTITY nrf_cyclic_real_interpolation_table_expression

An **nrf_cyclic_real_interpolation_table_expression** is a type of nrf_real_interpolation_table_expression that specifies a prescription defined by an interpolation table containing the dependent values for a single discrete basis of an independent parameter and a cyclic interpolation method.

Express specification:

```
ENTITY nrf_cyclic_real_interpolation_table_expression
  SUBTYPE OF(nrf_real_interpolation_table_expression);
  period : REAL;
WHERE
  has_one_independent_parameter: SIZEOF(independent_parameters) = 1;
END_ENTITY;
```

Attribute definitions:

— period specifies the period for the cyclic interpolation. Its quantity type is by definition the same as that of the first independent_quantity_types of the referenced interpolation_table.

Formal propositions:

— has_one_independent_parameter: There shall be only one indepent parameter.

4.2.6.17 FUNCTION nrf_verify_independent_quantity_types

The function **nrf_verify_independent_quantity_types** verifies that the quantity types of independent_parameters of an nrf_real_interpolation_table_expression and the independent_quantity_types of the associated interpolation_table are the same.

Express specification:

```
FUNCTION nrf_verify_independent_quantity_types(
     rite : nrf_real_interpolation_table_expression) : BOOLEAN;
  REPEAT i := 1 TO SIZEOF(rite.independent_parameters);
    IF rite.independent_parameters[i].quantity_type :<>:
        rite.interpolation_table.independent_quantity_types[i] THEN
      RETURN(FALSE);
```



```
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— rite specifies the candidate nrf_real_interpolation_table_expression to be verified.

4.2.6.18 ENTITY nrf_real_lookup_table

An **nrf_real_lookup_table** specifies a multi-dimensional mathematical space with discrete bases of independent quantity values that span a grid, and discrete sampled dependent quantity values at all grid points. It can be used to define an interpolation table.

Express specification:

```
ENTITY nrf_real_lookup_table;
  id                          : nrf_identifier;
  description                 : nrf_text;
  independent_quantity_types  : LIST [1:?] OF nrf_real_quantity_type;
  dependent_quantity_type     : nrf_real_quantity_type;
  independent_values          : LIST [1:?] OF LIST [1:?] OF REAL;
  dependent_values            : LIST [1:?] OF REAL;
WHERE
  has_correct_number_of_independent_quantities:
    SIZEOF(independent_quantity_types) = SIZEOF(independent_values);
  has_correct_number_of_dependent_quantities:
    nrf_verify_number_of_dependent_values_in_real_lookup_table(SELF);
END_ENTITY;
```

Attribute definitions:

— id specifies the identifier by which the dataspace is known.

— description specifies a textual description of the dataspace.

— independent_quantity_types specifies the list of independent quantity types for each of the bases of the dataspace.

— dependent_quantity_type specifies the quantity type of the dependent values in the dataspace.

— independent_values specifies the nested lists of discrete values that define each of the bases of the dataspace.

— dependent_values specifies the list of dependent values that populate the lookup table. The ordering of the values in dependent_values follows the ordering of the independent_values, in other words the index for the first basis of independent_values varies slowest and the index for the last basis of independent_values varies fastest.

EXAMPLE The code example below shows a possible implementation in EXPRESS to compute the index in the flat list of dependent_values from a given set of independent_values indices:

```
FUNCTION nrf_get_real_lookup_value_index(
    rlt : nrf_real_lookup_table;
    indices : LIST OF INTEGER) : INTEGER;
  LOCAL
    j : INTEGER;
  END_LOCAL;
  IF SIZEOF(rlt.independent_values) <> SIZEOF(indices) THEN
```



```
    RETURN(?);
  END_IF;
  j := 0;
  REPEAT i := 1 TO (SIZEOF(indices) - 1);
    j := (j + (indices[i] - 1)) * SIZEOF(rlt.independent_values[i+1]);
  END_REPEAT;
  j := j + indices[SIZEOF(indices)];
  RETURN(j);
END_FUNCTION;
```

Formal propositions:

- has_correct_number_of_independent_quantities: The number of independent_quantity_types shall be the same as the number of independent_values lists.
- has_correct_number_of_dependent_quantities: The number of dependent_values shall match the product of the numbers of independent_values along each of the dataspace's bases.

4.2.6.19 FUNCTION nrf_verify_number_of_dependent_values_in_real_lookup_table

The function **nrf_verify_number_of_dependent_values_in_real_lookup_table** verifies that the number of dependent_values for an nrf_real_lookup_table is correct.

Express specification:

```
FUNCTION nrf_verify_number_of_dependent_values_in_real_lookup_table(
     rlt : nrf_real_lookup_table) : BOOLEAN;
  LOCAL
    required_number_of_dependent_values : INTEGER := 1;
  END_LOCAL;
  REPEAT i := 1 TO SIZEOF(rlt.independent_values);
    required_number_of_dependent_values :=
      required_number_of_dependent_values * SIZEOF(rlt.independent_values[i]);
  END_REPEAT;
  IF SIZEOF(rlt.dependent_values) <> required_number_of_dependent_values THEN
    RETURN(FALSE);
  END_IF;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- rlt specifies the candidate nrf_real_lookup_table to be verified.

4.2.6.20 ENTITY nrf_integer_quantity_value_prescription

An **nrf_integer_quantity_value_prescription** is a type of nrf_any_quantity_value_prescription for an nrf_integer_quantity_type. It is an abstract supertype that provides a generic mechanism to reference an nrf_integer_quantity_literal or an nrf_integer_quantity_expression.

Express specification:

```
ENTITY nrf_integer_quantity_value_prescription
  ABSTRACT SUPERTYPE OF(ONEOF(
    nrf_integer_quantity_value_literal,
    nrf_integer_quantity_value_expression))
  SUBTYPE OF(nrf_any_quantity_value_prescription);
```

```
  SELF\nrf_any_quantity_value_prescription.quantity_type : nrf_integer_quantity_type;
DERIVE
  val : INTEGER := integer_values[1];
END_ENTITY;
```

Attribute definitions:

— val is derived to be the first integer value for the quantity_type, i.e. the best estimated value if quantity_type has an associated uncertainty_method. For an nrf_integer_quantity_value_literal it specifies the final value, for the nrf_integer_quantity_value_expression subtype it specifies the evaluated default value for the expression.

4.2.6.21 ENTITY nrf_integer_quantity_value_literal

An **nrf_integer_quantity_value_literal** is a type of nrf_integer_quantity_value_prescription that specifies a literal integer value for an nrf_integer_quantity_type.

Express specification:

```
ENTITY nrf_integer_quantity_value_literal
  SUBTYPE OF(nrf_integer_quantity_value_prescription);
END_ENTITY;
```

4.2.6.22 ENTITY nrf_integer_quantity_value_expression

An **nrf_integer_quantity_value_expression** is a type of nrf_integer_quantity_value_prescription that specifies the prescription of a value for an nrf_integer_quantity_type through an expression conforming to a given algorithmic language. The expression may reference one or more nrf_variable instances through their id.

Express specification:

```
ENTITY nrf_integer_quantity_value_expression
  SUBTYPE OF(nrf_integer_quantity_value_prescription);
  language     : nrf_algorithmic_language;
  expression   : nrf_algorithmic_expression;
  creator_tool : OPTIONAL nrf_tool_or_facility;
END_ENTITY;
```

Attribute definitions:

— language specifies the nrf_algorithmic_language in which the expression is expressed.

— expression specifies an algorithmic expression in a syntax that conforms to the algorithmic language specified in the language attribute.

— creator_tool optionally specifies the tool that was used in the creation of the expression.

4.2.6.23 ENTITY nrf_string_quantity_value_prescription

An **nrf_string_quantity_value_prescription** is a type of nrf_any_quantity_value_prescription for an nrf_string_quantity_type. It is an abstract supertype that provides a generic mechanism to reference an nrf_string_quantity_literal or an nrf_string_quantity_expression.

Express specification:


```
ENTITY nrf_string_quantity_value_prescription
  ABSTRACT SUPERTYPE OF(ONEOF(
    nrf_string_quantity_value_literal,
    nrf_string_quantity_value_expression))
  SUBTYPE OF(nrf_any_quantity_value_prescription);
  SELF\nrf_any_quantity_value_prescription.quantity_type : nrf_string_quantity_type;
DERIVE
  val : INTEGER := integer_values[1];
END_ENTITY;
```

Attribute definitions:

— val is derived to be the literal index into the string_values list of the quantity_type. For an nrf_string_quantity_value_literal it specifies the final value, for an nrf_string_quantity_value_expression it specifies the evaluated default value for the expression.

4.2.6.24 ENTITY nrf_string_quantity_value_literal

An **nrf_string_quantity_value_literal** is a type of nrf_string_quantity_value_prescription that specifies a literal string value for an nrf_string_quantity_type.

Express specification:

```
ENTITY nrf_string_quantity_value_literal
  SUBTYPE OF(nrf_string_quantity_value_prescription);
END_ENTITY;
```

4.2.6.25 ENTITY nrf_string_quantity_value_expression

An **nrf_string_quantity_value_expression** is a type of nrf_string_quantity_value_prescription that specifies the prescription of a value for an nrf_string_quantity_type through an expression conforming to a given algorithmic language. The expression may reference one or more nrf_variable instances through their id.

Express specification:

```
ENTITY nrf_string_quantity_value_expression
  SUBTYPE OF(nrf_string_quantity_value_prescription);
  language     : nrf_algorithmic_language;
  expression   : nrf_algorithmic_expression;
  creator_tool : OPTIONAL nrf_tool_or_facility;
END_ENTITY;
```

Attribute definitions:

— language specifies the nrf_algorithmic_language in which the expression is expressed.

— expression specifies an algorithmic expression in a syntax that conforms to the algorithmic language specified in the language attribute.

— creator_tool optionally specifies the tool that was used in the creation of the expression.

4.2.6.26 ENTITY nrf_enumeration_quantity_value_prescription

An **nrf_enumeration_quantity_value_prescription** is a type of nrf_any_quantity_value_prescription for an nrf_enumeration_quantity_type. It is an abstract supertype that provides a generic mechanism to reference an nrf_enumeration_quantity_literal or an nrf_enumeration_quantity_expression.

Express specification:

```
ENTITY nrf_enumeration_quantity_value_prescription
  ABSTRACT SUPERTYPE OF(ONEOF(
    nrf_enumeration_quantity_value_literal,
    nrf_enumeration_quantity_value_expression))
  SUBTYPE OF(nrf_any_quantity_value_prescription);
  SELF\nrf_any_quantity_value_prescription.quantity_type : nrf_enumeration_quantity_type;
DERIVE
  val : INTEGER := integer_values[1];
END_ENTITY;
```

Attribute definitions:

— val is derived to be the literal index into the enumeration_items list of the quantity_type. For an nrf_enumeration_quantity_value_literal it specifies the final value, for an nrf_enumeration_quantity_value_expression it specifies the evaluated default value for the expression.

4.2.6.27 ENTITY nrf_enumeration_quantity_value_literal

An **nrf_enumeration_quantity_value_literal** is a type of nrf_enumeration_quantity_value_prescription that specifies a literal enumeration value for an nrf_enumeration_quantity_type.

Express specification:

```
ENTITY nrf_enumeration_quantity_value_literal
  SUBTYPE OF(nrf_enumeration_quantity_value_prescription);
END_ENTITY;
```

4.2.6.28 ENTITY nrf_enumeration_quantity_value_expression

An **nrf_enumeration_quantity_value_expression** is a type of nrf_enumeration_quantity_value_prescription that specifies the prescription of a value for an nrf_enumeration_quantity_type through an expression conforming to a given algorithmic language. The expression may reference one or more nrf_variable instances through their id.

Express specification:

```
ENTITY nrf_enumeration_quantity_value_expression
  SUBTYPE OF(nrf_enumeration_quantity_value_prescription);
  language     : nrf_algorithmic_language;
  expression   : nrf_algorithmic_expression;
  creator_tool : OPTIONAL nrf_tool_or_facility;
END_ENTITY;
```

Attribute definitions:

- language specifies the nrf_algorithmic_language in which the expression is expressed.
- expression specifies an algorithmic expression in a syntax that conforms to the algorithmic language specified in the language attribute.
- creator_tool optionally specifies the tool that was used in the creation of the expression.

4.2.6.29 ENTITY nrf_tensor_quantity_value_prescription

An **nrf_tensor_quantity_value_prescription** is a type of nrf_any_quantity_value_prescription for an nrf_any_tensor_quantity_type. It is an abstract supertype that provides a generic mechanism to reference an nrf_any_tensor_quantity_literal or an nrf_any_tensor_quantity_expression.

Express specification:

```
ENTITY nrf_tensor_quantity_value_prescription
  ABSTRACT SUPERTYPE OF(ONEOF(
    nrf_tensor_quantity_value_literal,
    nrf_tensor_quantity_value_expression))
  SUBTYPE OF(nrf_any_quantity_value_prescription);
  SELF\nrf_any_quantity_value_prescription.quantity_type :
    nrf_any_tensor_quantity_type;
END_ENTITY;
```

4.2.6.30 ENTITY nrf_tensor_quantity_value_literal

An **nrf_tensor_quantity_value_literal** is a type of nrf_tensor_quantity_value_prescription that specifies literal values for an nrf_any_tensor_quantity_type.

Express specification:

```
ENTITY nrf_tensor_quantity_value_literal
  SUBTYPE OF(nrf_tensor_quantity_value_prescription);
END_ENTITY;
```

4.2.6.31 ENTITY nrf_tensor_quantity_value_expression

An **nrf_tensor_quantity_value_expression** is a type of nrf_tensor_quantity_value_prescription that specifies the prescription of the values for an nrf_any_tensor_quantity_type through an expression conforming to a given algorithmic language. The expression may reference one or more nrf_variable instances through their id.

Express specification:

```
ENTITY nrf_tensor_quantity_value_expression
  SUBTYPE OF(nrf_tensor_quantity_value_prescription);
  language     : nrf_algorithmic_language;
  expression   : nrf_algorithmic_expression;
  creator_tool : OPTIONAL nrf_label;
END_ENTITY;
```

Attribute definitions:

- language specifies the nrf_algorithmic_language in which the expression is expressed.

- expression specifies an algorithmic expression in a syntax that conforms to the algorithmic language specified in the language attribute.
- creator_tool optionally specifies the tool that was used in the creation of the expression.

4.2.6.32 ENTITY nrf_quantity_value_prescription_for_item

An **nrf_quantity_value_prescription_for_item** specifies an nrf_any_quantity_value_prescription for a given nrf_observable_item. It is used to prescribe the value for a quantity type that is associated with an nrf_observable_item. It can therefore be used in parametric model definitions.

Express specification:

```
ENTITY nrf_quantity_value_prescription_for_item;
  item         : nrf_observable_item;
  prescription : nrf_any_quantity_value_prescription;
END_ENTITY;
```

Attribute definitions:

- item specifies the nrf_observable_item for which the prescription is defined.
- prescription specifies the nrf_any_quantity_value_prescription.

4.2.7 NRF Network model representation UoF

The nrf_network_model_representation UoF collects the definitions needed for a generic representation of engineering objects as a hierarchical structure of models containing a network of discrete nodes and relationships between these nodes.

4.2.7.1 ENTITY nrf_observable_item

An **nrf_observable_item** is an abstract supertype that provides a generic mechanism to reference a subtype item for which a quantity can be observed. This is a lightweight entity without any name or other attributes so that it can be used in applications where there is a very large number of observable items.

Express specification:

```
ENTITY nrf_observable_item
  ABSTRACT SUPERTYPE OF (ONEOF(
    nrf_named_observable_item,
    nrf_observable_item_relationship(*,
    mgm_face*)));
END_ENTITY;
```

4.2.7.2 ENTITY nrf_observable_item_relationship

An **nrf_observable_item_relationship** is an abstract supertype that provides a generic mechanism to reference a subtype relationship between two or more instances of nrf_observable_item for which a quantity can be observed.

Express specification:


```
ENTITY nrf_observable_item_relationship
  (*ABSTRACT SUPERTYPE OF(ONEOF(mgm_face_pair))*)
  SUBTYPE OF(nrf_observable_item);
  items : LIST [2:?] OF nrf_observable_item;
END_ENTITY;
```

4.2.7.3 ENTITY nrf_named_observable_item_class

An **nrf_named_observable_item_class** is a specification of a class of nrf_named_observable_item instances.

The attribute name specifies the unique class name. It shall be defined in an external resource and is dependent on the analysis, simulation, test or operation discipline.

EXAMPLE 1 An nrf_network_model representing a thermal network model may be identified by an **nrf_named_observable_item_class** with name 'thermal_network_model'.

EXAMPLE 2 An nrf_network_node_relationship representing a TRI3 structural finite element may be identified by an **nrf_named_observable_item_class** with name 'structural_finite_element_tri3'.

Express specification:

```
ENTITY nrf_named_observable_item_class
  SUPERTYPE OF(ONEOF(
    nrf_network_model_class,
    nrf_network_node_class,
    nrf_network_node_relationship_class,
    nrf_material_class,
    nrf_named_observable_item_group_class(*,
    sma_celestial_body_class*)));
  name                          : nrf_non_blank_label;
  description                   : nrf_text;
  security_class                : OPTIONAL nrf_security_classification_level;
UNIQUE
  has_unique_name: name;
END_ENTITY;
```

Attribute definitions:

- name specifies the human-interpretable name of an instance of **nrf_named_observable_item_class**.
- description specifies the textual description of an instance of **nrf_named_observable_item_class**.
- security_class optionally specifies the security class for access to the data associated to the **nrf_named_observable_item_class**

Formal propositions:

- has_unique_name: name shall be unique for all **nrf_named_observable_item_class** instances in the dataset.

4.2.7.4 ENTITY nrf_named_observable_item

An **nrf_named_observable_item** is a type of nrf_observable_item with an individual identifier, name, description, classification and security_class. It is also a supertype that specializes into nrf_network_model, nrf_network_node, nrf_network_node_relationship, nrf_material and nrf_named_observable_item_group.

EXAMPLE A surface material can be represented as an **nrf_named_observable_item**, which has optical properties.

EXAMPLE A thermal network node can be represented as an **nrf_named_observable_item** with an observed temperature.

Express specification:

```
ENTITY nrf_named_observable_item
  SUPERTYPE OF(ONEOF(
    nrf_network_model,
    nrf_network_node,
    nrf_network_node_relationship,
    nrf_material,
    nrf_named_observable_item_group(*,
    sma_celestial_body*)))
  SUBTYPE OF(nrf_observable_item);
  id             : nrf_identifier;
  name           : nrf_label;
  description    : nrf_text;
  item_class     : nrf_named_observable_item_class;
  security_class : OPTIONAL nrf_security_classification_level;
END_ENTITY;
```

Attribute definitions:

- id specifies the identifier of an **nrf_named_observable_item**.
- name specifies the human-interpretable name of an **nrf_named_observable_item**.
- description specifies the textual description of an **nrf_named_observable_item**.
- item_class specifies the nrf_named_observable_item_class of which an **nrf_named_observable_item** is a member.
- security_class specifies the security class for access to the data associated to the **nrf_named_observable_item**.

4.2.7.5 ENTITY nrf_observable_item_list

An **nrf_observable_item_list** represents a list of nrf_observable_item instances. Such a list can be used as an item basis for an nrf_datacube.

Express specification:

```
ENTITY nrf_observable_item_list
  SUPERTYPE OF(ONEOF(nrf_named_observable_item_list));
  id             : nrf_identifier;
  name           : nrf_label;
  description    : nrf_text;
  security_class : OPTIONAL nrf_security_classification_level;
  items          : LIST [1:?] OF UNIQUE nrf_observable_item;
END_ENTITY;
```

Attribute definitions:

- id specifies the identifier of an instance of an **nrf_observable_item_list**.
- name specifies the human-interpretable name of an instance of **nrf_observable_item_list**.
- description specifies the textual description of an instance of **nrf_observable_item_list**.
- security_class optionally specifies the security class for access to the data associated to the **nrf_observable_item_list**
- items specifies the list of nrf_observable_item instances

4.2.7.6 ENTITY nrf_named_observable_item_list

An **nrf_named_observable_item_list** is a type of nrf_observable_item_list that specifies a list of nrf_named_observable_item instances. Such a list can be used as an item basis for an nrf_datacube.

Express specification:

```
ENTITY nrf_named_observable_item_list
  SUBTYPE OF(nrf_observable_item_list);
  SELF\nrf_observable_item_list.items : LIST [1:?] OF UNIQUE nrf_named_observable_item;
WHERE
  items_have_unique_identifiers:
    nrf_verify_unique_identifiers(items);
END_ENTITY;
```

Attribute definitions:

- items specifies the list of nrf_named_observable_item instances.

Formal propositions:

- items_have_unique_identifiers: All items shall have a unique id.

4.2.7.7 ENTITY nrf_named_observable_item_group_class

An **nrf_named_observable_item_group_class** specifies a class of nrf_named_observable_item_group instances, that is a named category of nrf_named_observable_item_group instances that share common characteristics and behaviour. The name of the class shall be defined in an external dictionary and is dependent on the analysis, simulation, test or operation discipline.

EXAMPLE An **nrf_named_observable_item_group_class** with name 'thermal_network_node_group_class' and as valid_item_classes a reference to an nrf_network_node_class with name 'thermal_network_node_class' can be defined to specify arbitrary groups of thermal network nodes for e.g. post-processing purposes.

Express specification:

```
ENTITY nrf_named_observable_item_group_class
  SUBTYPE OF(nrf_named_observable_item_class);
  valid_item_classes : SET OF nrf_named_observable_item_class;
END_ENTITY;
```

Attribute definitions:

— valid_item_classes specifies the set of nrf_named_observable_item_class instances that define the kind of nrf_named_observable_item instances that may be contained in an nrf_named_observable_item_group.

4.2.7.8 ENTITY nrf_named_observable_item_group

An **nrf_named_observable_item_group** specifies a group of nrf_named_observable_item instances defined within the scope of an nrf_network_model. Such a group can be used for any purpose in which is it useful to identify or treat a collection of named observable items together. Since an **nrf_named_observable_item_group** is itself an nrf_named_observable_item the specification of groups of groups is possible.

EXAMPLE Some typical applications for named groups of observable items are: treating a groups of analysis model nodes together for post-processing purposes or treating groups of geometric shapes together for interactive viewing purposes.

Express specification:

```
ENTITY nrf_named_observable_item_group
  SUBTYPE OF(nrf_named_observable_item);
  SELF\nrf_named_observable_item.item_class : nrf_named_observable_item_group_class;
  items : LIST OF UNIQUE nrf_named_observable_item;
INVERSE
  containing_model : nrf_network_model FOR groups;
UNIQUE
  has_unique_id_within_containing_model: containing_model, id;
WHERE
  items_have_valid_item_class:
    SIZEOF(QUERY(item <* items | item.item_class IN item_class.valid_item_classes)) =
    SIZEOF(items);
  items_belong_to_containing_model:
    nrf_verify_named_observable_items_in_model_tree(SELF);
  item_group_tree_is_acyclic:
    nrf_verify_acyclic_item_group_tree(SELF, [SELF]);
END_ENTITY;
```

Attribute definitions:

— item_class specifies the nrf_named_observable_item_group_class of which this group is a member.

— items specifies the list of nrf_named_observable_item instances that together form the group.

— containing_model specifies the nrf_network_model for which this group is defined.

Formal propositions:

— has_unique_id_within_containing_model: the id shall be unique within the nrf_network_model that contains the **nrf_named_observable_item_group**.

— items_have_valid_item_class: All items in the group have an item_class that is present in the valid_item_classes of the group's own item_class.

— items_belong_to_containing_model: all items shall belong to containing_model or its submodel tree

— item_group_tree_is_acyclic: the item groups form an acyclic tree, in other words there shall be no circular group references.

4.2.7.9 ENTITY nrf_network_model_class

An **nrf_network_model_class** is a specification of a class of nrf_network_model instances, that is a named category of nrf_network_model instances that share common characteristics and behaviour. The name of the class shall be defined in an external dictionary and is dependent on the analysis, simulation, test or operation discipline.

EXAMPLE An nrf_network_model representing a thermal network model can have a class with name 'thermal_network_model_class'.

EXAMPLE An nrf_network_model representing a structural finite element model can have a class with name 'structural_finite_element_model_class'.

Express specification:

```
ENTITY nrf_network_model_class
  SUBTYPE OF(nrf_named_observable_item_class);
  valid_node_classes              : SET OF nrf_network_node_class;
  valid_node_relationship_classes : SET OF nrf_network_node_relationship_class;
END_ENTITY;
```

Attribute definitions:

- valid_node_classes specifies the nrf_network_node_class instances that are allowed to be used by the nrf_network_node instances referenced by an nrf_network_model instance using an **nrf_network_model_class**.
- valid_node_relationship_classes specifies the nrf_network_node_relationship_class instances that are allowed to be used by the nrf_network_node_relationship instances referenced by an nrf_network_model instance using an **nrf_network_model_class**.

4.2.7.10 ENTITY nrf_network_model

An **nrf_network_model** specifies the representation of a product, and possibly of its environment, in the form of a network topology. The model is intended for use in analysis, simulation, test or operation activities. It provides a generic hierarchical decomposition structure of identified discrete observable items and the relationships between them. Any number of properties can be associated to any item in this structure, such as: (1) characteristic properties – for example material properties; (2) intrinsic properties – for example dimensions or mass; (3) predicted properties – for example analysis results; (4) assigned properties – for example settings for test parameters; (5) observed properties – for example sensor readings obtained in a test or in operation; (6) derived properties – for example statistics derived from sensor readings or analysis results. Through an nrf_model_represents_product_relationship it is possible to express the association between an nrf_product_definition that identifies a product (or part of a product) and the **nrf_network_model** that represents it.

EXAMPLE 1 A finite difference / lumped parameter thermal network model could be an **nrf_network_model** with thermal network nodes represented by nrf_network_node instances and thermal network conductors represented by nrf_network_node_relationship instances. There would be an nrf_network_node_relationship_class for each supported conductor type, such as *linear conductor*, *radiative coupling*, *one way linear conductor*, etc.

EXAMPLE 2 A structural finite element model could be an **nrf_network_model** with the finite element nodes represented by nrf_network_node instances and finite elements themselves represented by

nrf_network_node_relationship instances. There would be an nrf_network_node_relationship_class for each supported finite element type, such as *TRI3*, *QUAD4*, etc.

EXAMPLE 3 A test article could be represented by an **nrf_network_model** with the sensors or channels represented by nrf_network_node instances. There would be an nrf_network_node_class for each supported class of sensor, such as *strain gauge*, *electrical current meter*, *thermistor*, etc.

Express specification:

```
ENTITY nrf_network_model
  (*SUPERTYPE OF(ONEOF(mgm_meshed_geometric_model))*)
  SUBTYPE OF(nrf_named_observable_item);
  SELF\nrf_named_observable_item.item_class : nrf_network_model_class;
  base_model_id                      : OPTIONAL nrf_identifier;
  nodes                              : LIST OF UNIQUE nrf_network_node;
  node_relationships                 : LIST OF UNIQUE nrf_network_node_relationship;
  submodels                          : LIST OF UNIQUE nrf_network_model;
  groups                             : LIST OF UNIQUE nrf_named_observable_item_group;
  functions                          : LIST OF UNIQUE nrf_model_function;
  initializations                    : LIST OF UNIQUE nrf_datacube;
  variables                          : LIST OF UNIQUE nrf_variable;
  prescriptions                      : LIST OF UNIQUE
                                       nrf_quantity_value_prescription_for_item;
  constraints                        : LIST OF UNIQUE nrf_model_constraint;
  materials                          : LIST OF UNIQUE nrf_material;
  material_property_environment_list : OPTIONAL nrf_state_list;
  material_properties                : LIST OF LIST OF LIST OF UNIQUE
                                       nrf_any_quantity_value_prescription;
INVERSE
  containing_model : SET [0:1] OF nrf_network_model FOR submodels;
UNIQUE
  has_unique_id_within_containing_model: containing_model, id, item_class;
WHERE
  has_same_item_class_as_containing_model:
    nrf_verify_same_item_class_as_containing_model(SELF);
  nodes_have_valid_node_classes:
    nrf_verify_nodes_in_network_model(SELF);
  node_relationships_have_valid_node_relationship_classes:
    nrf_verify_node_relationships_in_network_model(SELF);
  submodel_tree_is_acyclic:
    nrf_verify_acyclic_network_model_tree(SELF, [SELF]);
  nodes_referenced_in_relationships_are_in_model_tree:
    nrf_verify_nodes_referenced_in_relationships(SELF);
  has_complete_list_of_material_properties:
    nrf_verify_complete_list_of_material_properties(SELF);
  material_property_environment_quantity_type_is_scalar:
    (NOT EXISTS(material_property_environment_list)) OR
    ('NRF_ARM.NRF_ANY_SCALAR_QUANTITY_TYPE'
      IN TYPEOF(material_property_environment_list.quantity_type));
  has_same_material_property_environment_as_containing_model:
    nrf_verify_same_material_property_environment_as_containing_model(SELF);
END_ENTITY;
```

Attribute definitions:

— item_class specifies the nrf_network_model_class identifying the class of an **nrf_network_model**.

— base_model_id optionally specifies the identifier of a common base model from which an **nrf_network_model** instance is derived. It can be used by receiving processors to reconstruct the master definition of an **nrf_network_model** from multiple **nrf_network_model** occurrences (with the same base_model_id) that share a common definition.

- nodes specifies the list of nodes that are part of an **nrf_network_model**. The nodes are represented by nrf_network_node instances.
- node_relationships specifies the list of node- relationships that are part of an **nrf_network_model**. The node-relationships are represented by nrf_network_node_relationship instances
- submodels specifies the list of submodels that are part of an **nrf_network_model**. The submodels themselves are other **nrf_network_model** instances. Through this composition relationship a hierarchical model/submodel tree can be created – a parent model is the "whole" and its submodels are the "parts". The purpose of such a model/submodel tree is to support a hierarchical, modular breakdown for nrf_network models.
- groups specifies named observable item groups of items that are part of the model. See a more elaborate definition in nrf_named_observable_item_group.
- functions specifies a list of user-defined functions that are part of a (analysis or simulation) model specification. This functions may be called in expressions defined in prescriptions and in the execution script that governs execution of a case (see nrf_case) for the model.
- initializations specifies a list of nrf_datacube instances that assign values to quantity types for items that are part of the model. This is in particular appropriate and efficient for the bulk of the literal assignments. These datacubes shall only know one state: the ' initial' state.
- variables specifies a list of named variables that may be used in the case, in particular inside expressions defined in prescriptions.
- prescriptions specifies a list of value prescriptions for quantities of items that are part of the model, including the definition of boundary conditions.
- constraints specifies a list of user-defined constraints that are part of a (analysis or simulation) model specification.
- materials specifies the materials defined for an **nrf_network_model**.
- material_property_environment_list optionally specifies the list of material property environments for which material properties can be defined. Each state in the associated nrf_state_list represents a discrete environment for which a collection of material property values are defined.
- material_properties specifies a nested list of nrf_any_quantity_value_prescription instances that define the properties for each of the materials for each of the material property environments. In other words the outer list loops over the material property environments, the middle list loops over the materials, and the inner list loops over the quantity value prescriptions.
- containing_model specifies the **nrf_network_model** of which this **nrf_network_model** is a submodel, if any.

Formal propositions:

- has_unique_id_within_containing_model: The combination of id and item_class shall be unique within the **nrf_network_model** that contains this **nrf_network_model** as a submodel, in other words submodels of the same kind shall have unique identifiers. Be aware that proposition has_same_item_class_as_containing_model below requires that all submodels are of the same item_class. The end result is that only root models can have duplicate identifiers for models of different item_class.
- has_same_item_class_as_containing_model: The item_class shall be the same as the one of the containing_model, effectively asserting that all models in a submodel tree have the same item_class.
- nodes_have_valid_node_classes: The nrf_network_node_class of all the nrf_network_nodes in nodes shall be one of the nrf_network_node_class instances specified by model_class through valid_node_classes.

- node_relationships_have_valid_node_relationship_classes: The nrf_network_node_relationship class of all the nrf_network_node-relationships in node_relationships shall be one of the nrf_network_node_relationship_class instances specified by the model_class through valid_node_relationship_classes.
- submodel_tree_is_acyclic: The submodel tree shall form an acyclic graph, that is it shall not contain any circular **nrf_network_model** references.
- nodes_referenced_in_relationships_are_in_model_tree: The submodels referenced through a nrf_submodel_node_reference shall be in the submodel tree of an **nrf_network_model**.
- has_complete_list_of_material_properties: The material_properties shall specify a complete list. This implies the following: (a) The material_property_environment_list either does not exist and the material_properties list is empty or the size of the outer list of material_properties shall be equal to the number of states (that represent discrete environments) defined in the material_property_environment_list; (b) The size of each of the second level lists of material_properties shall be equal to the number of materials; (c) Material properties shall be specified in material_properties for each of the required_quantity_type_names of the item_class of each material in materials.
- material_property_environment_quantity_type_is_scalar: If it exists the quantity_type of the material_property_environment_list shall be a scalar quantity type.
- has_same_material_property_environment_as_containing_model: The material_property_environment_list shall be the same as the one in its containing_model, effectively asserting that all models in a submodel tree use the same material_property_environment_list.

4.2.7.11 FUNCTION nrf_verify_named_observable_items_in_model_tree

The function **nrf_verify_named_observable_items_in_model_tree** verifies that all items in an nrf_named_observable_item_group belong to the containing_model of the group. The function returns TRUE when this is the case and FALSE otherwise.

Express specification:

```
FUNCTION nrf_verify_named_observable_items_in_model_tree(
  noig : nrf_named_observable_item_group) : BOOLEAN;
  REPEAT i := 1 TO SIZEOF(noig.items);
    IF NOT (nrf_verify_item_in_model_tree(
        noig.items[i], noig.containing_model)) THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- noig specifies the nrf_named_observable_item_group that is to be verified.

4.2.7.12 FUNCTION nrf_verify_acyclic_item_group_tree

The function **nrf_verify_acyclic_item_group_tree** recursively verifies that the possible hierarchical specification of groups in an nrf_named_observable_item_group form an acyclic graph, in other words there are no circular references. The function returns TRUE when this is the case and FALSE otherwise.

Express specification:

```
FUNCTION nrf_verify_acyclic_item_group_tree(
  noig : nrf_named_observable_item_group;
  supergroups : LIST OF nrf_named_observable_item_group) : BOOLEAN;
  REPEAT i := 1 TO SIZEOF(noig.items);
    IF 'NRF_ARM.NRF_NAMED_OBSERVABLE_ITEM_GROUP' IN TYPEOF(noig.items[i]) THEN
      IF noig.items[i] IN supergroups THEN
        RETURN(FALSE);
      END_IF;
      IF NOT nrf_verify_acyclic_item_group_tree(
          noig.items[i], supergroups + noig.items[i]) THEN
        RETURN(FALSE);
      END_IF;
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— noig specifies the nrf_named_observable_item_group that is to be verified.

— supergroups specifies the list of nrf_named_observable_item_group instances that are at the level of noig or higher up the tree.

4.2.7.13 FUNCTION nrf_verify_nodes_in_network_model

The function **nrf_verify_nodes_in_network_model** verifies that all nrf_network_node instances specified in an nrf_network_model belong to a valid nrf_network_model_class. The function returns TRUE when this is the case and FALSE otherwise.

Each nrf_network_node has an associated nrf_network_node_class. Each nrf_network_model has an associated nrf_network_model_class, that specifies the valid node classes through its valid_node_classes attribute.

Express specification:

```
FUNCTION nrf_verify_nodes_in_network_model(
    a_model : nrf_network_model) : BOOLEAN;
  LOCAL
    node : nrf_network_node;
  END_LOCAL;
  REPEAT i := 1 TO SIZEOF(a_model.nodes);
    node := a_model.nodes[i];
    IF NOT (node.item_class IN a_model.item_class.valid_node_classes) THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— a_model specifies the candidate nrf_network_model that is to be verified.

4.2.7.14 FUNCTION nrf_verify_node_relationships_in_network_model

The function **nrf_verify_node_relationships_in_network_model** verifies that all nrf_network_node_relationship instances specified in an nrf_network_model belong to a valid nrf_network_model_relationship_class. The function returns TRUE when this is the case and FALSE otherwise.

Each nrf_network_node_relationship has an associated nrf_network_node_relationship_class. Each nrf_network_model has an associated nrf_network_model_class, that specifies the valid node relationship classes through its valid_node_relationship_classes attribute.

Express specification:

```
FUNCTION nrf_verify_node_relationships_in_network_model(
    a_model : nrf_network_model): BOOLEAN;
  LOCAL
    node_relationship : nrf_network_node_relationship;
    model_class : nrf_network_model_class;
  END_LOCAL;
  model_class := a_model.item_class;
  REPEAT i := 1 TO SIZEOF(a_model.node_relationships);
    node_relationship := a_model.node_relationships[i];
    IF NOT (node_relationship.item_class
        IN model_class.valid_node_relationship_classes) THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— a_model specifies the candidate nrf_network_model that is to be verified.

4.2.7.15 FUNCTION nrf_verify_acyclic_network_model_tree

The function **nrf_verify_acyclic_network_model_tree** verifies that there is no circular reference in the submodel tree of a nrf_network_model. In other words, the submodel tree forms an acyclic graph. The function returns TRUE if this is the case and FALSE otherwise.

Express specification:

```
FUNCTION nrf_verify_acyclic_network_model_tree(
    a_model: nrf_network_model;
    supermodels : LIST OF nrf_network_model) : BOOLEAN;

  REPEAT i := 1 TO SIZEOF(a_model.submodels);
    IF (a_model.submodels[i] IN supermodels) THEN
      RETURN(FALSE);
    END_IF;
    IF NOT nrf_verify_acyclic_network_model_tree(
        a_model.submodels[i], supermodels + a_model.submodels) THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```


Argument definitions:

— a_model specifies the candidate nrf_network_model that is to be verified.

— supermodels specifies the list of nrf_network_model instances that occur at the same or higher level as a_model.

4.2.7.16 FUNCTION nrf_verify_nodes_referenced_in_relationships

The function **nrf_verify_nodes_referenced_in_relationships** verifies that all node_relationships in an nrf_network_model only reference nodes that are part of the model itself or one of its submodels. The function returns TRUE if this is the case and FALSE otherwise.

Express specification:

```
FUNCTION nrf_verify_nodes_referenced_in_relationships(
    a_model : nrf_network_model): BOOLEAN;
  LOCAL
    node : nrf_network_node;
    relationship : nrf_network_node_relationship;
  END_LOCAL;
  REPEAT i := 1 TO SIZEOF(a_model.node_relationships);
    relationship := a_model.node_relationships[i];
    REPEAT j := 1 TO SIZEOF(relationship.related_nodes);
      node := relationship.related_nodes[j];
      IF (NOT(node IN a_model.nodes)) THEN
        IF (NOT(nrf_verify_node_in_submodel_tree(a_model, node))) THEN
          RETURN(FALSE);
        END_IF;
      END_IF;
    END_REPEAT;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— a_model specifies the candidate nrf_network_model that is to be verified.

4.2.7.17 FUNCTION nrf_verify_node_in_submodel_tree

The FUNCTION **nrf_verify_node_in_submodel_tree** verifies that an nrf_network_node is defined in one of the submodels of an nrf_network_model. The FUNCTION returns TRUE if the submodel is found in the submodel tree of the model. Otherwise it returns FALSE.

Express specification:

```
FUNCTION nrf_verify_node_in_submodel_tree(
    a_model: nrf_network_model;
    a_node : nrf_network_node): BOOLEAN;
  LOCAL
    submodel : nrf_network_model;
  END_LOCAL;
  REPEAT i := 1 TO SIZEOF(a_model.submodels);
    submodel := a_model.submodels[i];
    IF (a_node IN submodel.nodes) THEN
      RETURN(TRUE);
    ELSE
```



```
      IF (nrf_verify_node_in_submodel_tree(submodel, a_node)) THEN
        RETURN(TRUE);
      END_IF;
    END_IF;
  END_REPEAT;
  RETURN(FALSE);
END_FUNCTION;
```

Argument definitions:

- a_model specifies the candidate nrf_network_model that is to be verified.
- a_node specifies the candidate nrf_network_node that is to be found in the submodel tree of a_model.

4.2.7.18 FUNCTION nrf_verify_complete_list_of_material_properties

The function **nrf_verify_complete_list_of_material_properties** verifies that the materials, the material_property_environment_list and the material_properties of a given nrf_network_model together specify a complete list of material properties or none at all. The function returns TRUE when this is the case and FALSE otherwise.

Express specification:

```
FUNCTION nrf_verify_complete_list_of_material_properties(
    a_model : nrf_network_model) : BOOLEAN;
  LOCAL
    required_qt_names : LIST OF nrf_non_blank_label;
    found : BOOLEAN;
  END_LOCAL;

  IF NOT EXISTS(a_model.material_property_environment_list) THEN
    IF SIZEOF(a_model.material_properties) = 0 THEN
      RETURN(TRUE);
    ELSE
      RETURN(FALSE);
    END_IF;
  END_IF;

  IF a_model.material_property_environment_list.number_of_states <>
      SIZEOF(a_model.material_properties) THEN
    RETURN(FALSE);
  END_IF;

  REPEAT i_mat := 1 TO SIZEOF(a_model.materials);
    required_qt_names :=
      a_model.materials[i_mat].item_class.all_required_quantity_type_names;
    REPEAT i_mpe := 1 TO SIZEOF(a_model.material_properties);
      IF SIZEOF(a_model.material_properties[i_mpe]) <> SIZEOF(a_model.materials) THEN
        RETURN(FALSE);
      END_IF;
      IF SIZEOF(a_model.material_properties[i_mpe][i_mat]) <>
          SIZEOF(required_qt_names) THEN
        RETURN(FALSE);
      END_IF;
      REPEAT i_rqtn := 1 TO SIZEOF(required_qt_names);
        found := FALSE;
        REPEAT i_prop := 1 TO SIZEOF(a_model.material_properties[i_mpe][i_mat])
            WHILE NOT found;
          IF required_qt_names[i_rqtn] =
              a_model.material_properties[i_mpe][i_mat][i_prop].quantity_type.name THEN
```

```
            found := TRUE;
          END_IF;
        END_REPEAT;
        IF NOT found THEN
          RETURN(FALSE);
        END_IF;
      END_REPEAT;
    END_REPEAT;
  END_REPEAT;

  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— a_model specifies the nrf_network_model for which the material_properties list is to be verified.

4.2.7.19 FUNCTION nrf_verify_same_material_property_environment_as_containing_model

The function **nrf_verify_same_material_property_environment_as_containing_model** verifies that an nrf_network_model uses the same material_property_environment_list as its containing_model, if this exists. The function returns TRUE when this is the case and FALSE otherwise.

Express specification:

```
FUNCTION nrf_verify_same_material_property_environment_as_containing_model(
    a_model : nrf_network_model) : BOOLEAN;

  IF SIZEOF(a_model.containing_model) = 1 THEN
    IF EXISTS(a_model.material_property_environment_list) THEN
      IF EXISTS(a_model.containing_model[1].material_property_environment_list) THEN
        IF a_model.containing_model[1].material_property_environment_list :<>:
            a_model.material_property_environment_list THEN
          RETURN(FALSE);
        END_IF;
      ELSE
        RETURN(FALSE);
      END_IF;
    ELSE
      IF EXISTS(a_model.containing_model[1].material_property_environment_list) THEN
        RETURN(FALSE);
      END_IF;
    END_IF;
  END_IF;

  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— a_model specifies the nrf_network_model to be verified.

4.2.7.20 FUNCTION nrf_verify_same_item_class_as_containing_model

The function **nrf_verify_same_item_class_as_containing_model** verifies that an nrf_network_model uses the same item_class as its containing_model, if this exists. The function returns TRUE when this is the case and FALSE otherwise.

Express specification:

```
FUNCTION nrf_verify_same_item_class_as_containing_model(
    a_model : nrf_network_model) : BOOLEAN;

  IF SIZEOF(a_model.containing_model) = 1 THEN
    IF a_model.containing_model[1].item_class :<>:
        a_model.item_class THEN
      RETURN(FALSE);
    END_IF;
  END_IF;

  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— a_model specifies the nrf_network_model to be verified.

4.2.7.21 ENTITY nrf_network_node_class

An **nrf_network_node_class** is a specification of a class of nrf_network_node instances, that is: a named category of nodes which share common characteristics and behaviour.

Express specification:

```
ENTITY nrf_network_node_class
  SUBTYPE OF(nrf_named_observable_item_class);
END_ENTITY;
```

4.2.7.22 ENTITY nrf_network_node

An **nrf_network_node** is a type of nrf_named_observable_item that specifies an atomic item of discretization in an nrf_network_model. An **nrf_network_node** is a vertex in the network topology of an nrf_network_model. The node is used to represent a physical object or parts thereof. An **nrf_network_node** serves as an item for which a quantity can be observed.

Express specification:

```
ENTITY nrf_network_node
  (*SUPERTYPE OF(ONEOF(mgm_any_meshed_geometric_item))*)
  SUBTYPE OF(nrf_named_observable_item);
  SELF\nrf_named_observable_item.item_class : nrf_network_node_class;
INVERSE
  containing_model : nrf_network_model FOR nodes;
UNIQUE
  has_unique_id_within_containing_model: containing_model, id;
END_ENTITY;
```

Attribute definitions:

— item_class specifies the nrf_network_node_class identifiying the class of an **nrf_network_node**

— containing_model specifies the nrf_network_model that contains an **nrf_network_node**.

Formal propositions:

— has_unique_id_within_containing_model: the id shall be unique within the nrf_network_model that contains the **nrf_network_node**.

4.2.7.23 ENTITY nrf_network_node_relationship_class

An **nrf_network_node_relationship_class** is a specification of a class of nrf_network_node_relationship instances, that is: a named category of nrf_network_node_relationship instances which share common characteristics and behaviour.

Express specification:

```
ENTITY nrf_network_node_relationship_class
  SUBTYPE OF(nrf_named_observable_item_class);
END_ENTITY;
```

4.2.7.24 ENTITY nrf_network_node_relationship

An **nrf_network_node_relationship** is a type of nrf_named_observable_item that specifies a relationship between two or more nrf_network_node instances of an nrf_network_model. An **nrf_network_node_relationship** defines one or more edges in the network topology of an nrf_network_model. The **nrf_network_node_relationship** is an item for which a quantity can be observed.

EXAMPLE Examples of such relationships are: the thermal radiative exchange factors between two thermal-radiative faces; the linear conductors, radiative couplings and mass flow links between two thermal network nodes; a finite element defined as a relationship between its vertex nodes.

Express specification:

```
ENTITY nrf_network_node_relationship
  SUBTYPE OF(nrf_named_observable_item);
  SELF\nrf_named_observable_item.item_class : nrf_network_node_relationship_class;
  related_nodes    : LIST [2:?] OF nrf_network_node;
INVERSE
  containing_model : nrf_network_model FOR node_relationships;
UNIQUE
  has_unique_id_within_containing_model: containing_model, id;
END_ENTITY;
```

Attribute definitions:

— item_class specifies the nrf_network_node_relationship_class identifiying the class of an **nrf_network_node_relationship**.

— related_nodes specifies the nrf_network_node instances that are related in an **nrf_network_node_relationship**. The significance of the order in which the nodes appear in the list depends on the given item_class and is defined in the

— containing_model specifies the nrf_network_model that contains an **nrf_network_node_relationship**.

NOTE One of the WHERE rules of the nrf_network_model guarantees that all nrf_network_node instances that are referenced in nodes are part of the model that contains an **nrf_network_node_relationship** or one of its submodels. In other words an **nrf_network_node_relationship** may only reference nodes downwards in its model/submodel tree, and may not reference any nodes upwards in its parent model tree.

Formal propositions:

— has_unique_id_within_containing_model: the id shall be unique within nrf_network_model that contains the **nrf_network_node_relationship**.

4.2.7.25 ENTITY nrf_model_represents_product_relationship

An **nrf_model_represents_product_relationship** is a relationship between an nrf_network_model and an nrf_product_definition that identifies the product (or part of a product) that is represented by the model representation.

Express specification:

```
ENTITY nrf_model_represents_product_relationship;
  model_representation : nrf_network_model;
  represented_product  : nrf_product_definition;
END_ENTITY;
```

Attribute definitions:

— model_representation specifies the nrf_network_model representing a product.

— represented_product specifies the nrf_product_definition defining a product.

4.2.8 NRF Cases, runs and results UoF

This UoF collects the application objects needed for the identification and definition of analysis, simulation, test or operation cases and runs and the results data they produce.

The results data are stored as properties of observable items for a certain state of a system that is being observed. As defined in UoF nrf_network_model, observable items can be structured in network model topologies to represent a system. The leading principle is that all properties can be captured in a so-called *datacube*, which is a mathematical space that has the following three dimensions: an observable item basis, a quantity type basis, and a state basis.

A property of an observable item is then defined as the value for a given quantity type and a given (discrete) state of the system. As defined in UoF nrf_quantities_and_units, quantity types can be scalars, vectors or higher order tensors.

4.2.8.1 ENTITY nrf_root

An **nrf_root** specifies the central entry of a Network-model Results Format dataset. It collects global meta-data and the references to all top level instances that are the starting points to navigate the dataset. There is only one **nrf_root** instance in a dataset (see RULE nrf_root_is_singleton).

Express specification:

```
ENTITY nrf_root;
  id                              : nrf_identifier;
  name                            : nrf_label;
  description                     : nrf_text;
  project                         : nrf_organizational_project;
  schema_uri                      : nrf_uniform_resource_identifier;
  dictionary_uri_list             : LIST OF nrf_uniform_resource_identifier;
```



```
  conformance_class                   : nrf_positive_integer;
  creation_date_and_time              : nrf_date_and_time ;
  last_modification_date_and_time     : OPTIONAL nrf_date_and_time;
  contact_person_organization         : nrf_person_and_organization;
  approvals                           : LIST OF nrf_approval;
  security_class                      : OPTIONAL nrf_security_classification_level;
  undefined_real_sentinel             : REAL;
  positive_infinity_real_sentinel     : REAL;
  negative_infinity_real_sentinel     : REAL;
  undefined_integer_sentinel          : INTEGER;
  positive_infinity_integer_sentinel  : INTEGER;
  negative_infinity_integer_sentinel  : INTEGER;
  root_models                         : LIST OF UNIQUE nrf_network_model;
  root_cases                          : LIST OF UNIQUE nrf_case;
WHERE
  has_valid_root_case_models:
    SIZEOF(root_cases) = SIZEOF(QUERY(a_case <* root_cases |
    a_case.for_model IN root_models));
END_ENTITY;
```

Attribute definitions:

- id specifies the identifier of an instance of **nrf_root**.
- name specifies the human-interpretable name of an instance of **nrf_root**.
- description specifies the textual description of an instance of **nrf_root**.
- project specifies a project to which an **nrf_root** pertains.
- schema_uri specifies the URI of the EXPRESS schema that was used to create the dataset
- dictionary_uri_list specifies the URI(s) of the NRF dictionaries that were used to create the dataset. Typically one standard dictionary is used, which is identified by the first URI in the list. The standard dictionary can be extended by additional predefined instances. The extended dictionary would be identified by the second URI in the list and so forth. Each extended dictionary contains a full copy of the instances of its base dictionary. This list can be used to trace the origins of the current dictionary.
- conformance_class specifies the conformance class identifier as defined in clause 5
- creation_date_and_time specifies the date and time of creation of the dataset
- last_modification_date_and_time optionally specifies the date and time of last modification of the dataset
- contact_person_organization specifies the contact person and organization that created the dataset
- approvals specifies the nrf_approval instances applicable to the dataset
- security_class optionally specifies security class for the information in the dataset
- undefined_real_sentinel specifies the value representing the undefined real value in the dataset
- positive_infinity_real_sentinel specifies the value representing the positive infinite real in the dataset
- negative_infinity_real_sentinel specifies the value representing the negative infinite real in the dataset
- undefined_integer_sentinel specifies the value representing undefined integer value in the dataset
- positive_infinity_integer_sentinel specifies the value representing the positive infinite integer in the dataset
- negative_infinity_integer_sentinel specifies the value representing the negative infinite integer in the dataset

— root_models specifies the root models in the dataset.
— root_cases specifies the root cases for the nrf_network_model instances in root_models.

Formal propositions:

— has_valid_root_case_models: Each nrf_network_model used by any of the nrf_case instances in root_cases must be a root model, in other words such a model must be registered in root_models.

4.2.8.2 ENTITY nrf_case

An **nrf_case** specifies the identification, description and definition of an analysis, simulation, test or operation case for a given nrf_network_model. It may specify one or more initializations, variables, prescriptions, named events and named intervals. An **nrf_case** may be decomposed into a sequence of one or more subcases. The named events may be used to provide sequencing and synchronization points in the timeline of a case.

NOTE In existing engineering analysis or simulation tools the distinction between what belongs to *model* and what belongs to *case* is often not so clear, there is often a grey area of overlap, or the distinction is simply not made and everything is called *model*. This is not considered a problem since on export a STEP-NRF processor can map the ensemble of nrf_network_model and **nrf_case** to as many of such *integrated models* as there are nrf_network_model and **nrf_case** combinations and on import a STEP-NRF processor can map such an *integrated model* to a comprehensive nrf_network_model and a single relatively empty **nrf_case**.

Express specification:

```
ENTITY nrf_case
  (*SUPERTYPE OF(ONEOF(sma_space_mission_case))*);
  id                   : nrf_identifier;
  name                 : nrf_label;
  description          : nrf_text;
  base_case_id         : OPTIONAL nrf_identifier;
  for_model            : nrf_network_model;
  state_quantity_type  : nrf_any_scalar_quantity_type;
  initializations      : LIST OF UNIQUE nrf_datacube;
  variables            : LIST OF UNIQUE nrf_variable;
  prescriptions        : LIST OF UNIQUE nrf_quantity_value_prescription_for_item;
  constraints          : LIST OF UNIQUE nrf_model_constraint;
  events               : LIST OF UNIQUE nrf_case_event;
  intervals            : LIST OF UNIQUE nrf_case_interval;
  subcases             : LIST OF UNIQUE nrf_case;
INVERSE
  runs                 : SET OF nrf_run FOR for_case;
UNIQUE
  has_unique_id_for_model: for_model, id;
WHERE
  has_valid_items_in_initializations:
    nrf_verify_initializations(SELF);
  subcases_reference_same_model:
    SIZEOF(QUERY(sc <* subcases | sc.for_model :=: for_model)) = SIZEOF(subcases);
  case_tree_is_acyclic:
    nrf_verify_acyclic_case_tree(SELF, [SELF]);
  events_have_unique_identifiers:
    nrf_verify_unique_identifiers(events);
  intervals_have_unique_identifiers:
    nrf_verify_unique_identifiers(intervals);
END_ENTITY;
```

Attribute definitions:

- id specifies the identifier of an instance of **nrf_case**.
- name specifies the human-interpretable name of an instance of **nrf_case**.
- description specifies the textual description of an instance of **nrf_case**.
- base_case_id optionally specifies the identifier of a common base case from which an **nrf_case** instance is derived. It can be used by receiving processors to reconstruct the master definition of an **nrf_case** from multiple **nrf_case** occurrences (with the same base_case_id) that share a common definition.
- for_model specifies the nrf_network_model for which an **nrf_case** is defined.
- state_quantity_type specifies the default quantity type that is used to identify discrete states during a run of the case, typically this would be a 'time' or 'frequence' quantity type.
- initializations specifies a list of nrf_datacube instances that assign values to quantity types for items that are part of the model that is associated with the case (through attribute for_model). This is in particular appropriate and efficient for the bulk of the literal assignments. These initializations may override initializations from the nrf_network_model associated through for_model.
- variables specifies a list of named variables that may be used in the case, in particular inside expressions defined in prescriptions. These variables may override variables from the nrf_network_model associated through for_model.
- prescriptions specifies a list of value prescriptions for quantities of items that are part of the model associated with the case (through attribute for_model), including the definition of boundary conditions. These prescriptions may override prescriptions from the nrf_network_model associated through for_model.
- constraints specifies a list of user-defined constraints that are part of a (analysis or simulation) model specification. These contraints may override constraints from the nrf_network_model associated through for_model.
- events specifies the list of named events defined for the case. If the list contains only one event it shall be considered the start event. If the list contains two or more events the first event shall be the start event and the last event shall be the end event.
- intervals specifies the list of named intervals defined for the case.
- subcases specifies a sequence of next lower level **nrf_case** instances that constitute the current **nrf_case**. Synchronization between case and subcases can be specified by referencing common nrf_event instances.
- runs specifies a set of nrf_run instances containing information produced during the execution of an **nrf_case**.

Formal propositions:

- has_unique_id_for_model: The id shall be unique for the associated nrf_network_model.
- has_valid_items_in_initializations: All nrf_observable_items referenced in the item_basis of the nrf_datacube instances referenced in initializations shall belong to the model/submodel tree defined by for_model.
- subcases_reference_same_model: All subcases shall reference the same nrf_network_model as the parent **nrf_case**.
- case_tree_is_acyclic: The subcase tree shall form an acyclic graph, that is it shall not contain any circular **nrf_case** references.
- events_have_unique_identifiers: The identifiers of the events shall be unique.

— intervals_have_unique_identifiers: The identifiers of the intervals shall be unique.

4.2.8.3 ENTITY nrf_case_event

An **nrf_case_event** specifies an identified event for use in an nrf_case. Such events are used to provide sequencing and synchronization points in the timeline for possible subcases associated with an nrf_case.

Express specification:

```
ENTITY nrf_case_event;
  id          : nrf_identifier;
  name        : nrf_label;
  description : nrf_text;
  state       : OPTIONAL nrf_any_quantity_value_prescription;
END_ENTITY;
```

Attribute definitions:

— id specifies the identifier of an instance of **nrf_case_event**.

— name specifies the human-interpretable name of an instance of **nrf_case_event**.

— description specifies the textual description of an instance of **nrf_case_event**.

— state optionally specifies a state quantity value at which the event occurs

4.2.8.4 ENTITY nrf_case_interval

An **nrf_case_interval** specifies an identified interval for use in an nrf_case. Such intervals are used to provide identification of the time interval between two events.

Express specification:

```
ENTITY nrf_case_interval;
  id          : nrf_identifier;
  name        : nrf_label;
  description : nrf_text;
  start_event : nrf_case_event;
  end_event   : nrf_case_event;
END_ENTITY;
```

Attribute definitions:

— id specifies the identifier of an instance of **nrf_case_interval**.

— name specifies the human-interpretable name of an instance of **nrf_case_interval**.

— description specifies the textual description of an instance of **nrf_case_interval**.

— start_event specifies the nrf_case_event at which the interval starts.

— end_event specifies the nrf_case_event at which the interval ends.

4.2.8.5 ENTITY nrf_run

An **nrf_run** specifies a container holding the meta-data and results of an executed analysis, simulation, test or operation run. It also references the nrf_case with its associated nrf_network_model that was used to execute the run.

NOTE It is desirable to create globally unique identifiers for runs, so that run results can always be distinguished and traced. A practical scheme to achieve this without the need to resort to a central run id issuing mechanism would be to assign a string in the following format *'hostname.yyyy-mm- dd*T*hh*:*mm*:*ss*.*sss*Z', i.e. the hostname of the machine on which the run was produced concatenated with the start date-and-timestamp in milliseconds in ISO 8601 UTC format.

Express specification:

```
ENTITY nrf_run;
  id                        : nrf_identifier;
  name                      : nrf_label;
  description               : nrf_text;
  for_case                  : nrf_case;
  start_timestamp           : nrf_date_and_time;
  end_timestamp             : OPTIONAL nrf_date_and_time;
  creator_tool_or_facility  : nrf_tool_or_facility;
  input_run_identifiers     : LIST OF nrf_identifier;
  results                   : LIST OF nrf_datacube;
UNIQUE
  has_unique_id: id;
END_ENTITY;
```

Attribute definitions:

- id specifies the identifier of an instance of **nrf_run**.
- name specifies the human-interpretable name of an instance of **nrf_run**.
- description specifies the textual description of an instance of **nrf_run**.
- for_case specifies the nrf_case for which the run was executed.
- start_timestamp specifies the date and time at which the execution of the run started.
- end_timestamp optionally specifies the date and time at which the execution of the run completed.
- creator_tool_or_facility specifies the tool or facility that was used to execute the run.
- input_run_identifiers specifies a list of id's of **nrf_run** instances that were used to produce the run. This allows for trace- back to preceeding runs (and their associated models and cases) without the need to keep all intermediate **nrf_run** instances. The identifiers should be ordered chronologically in order of creation, the oldest first.
- results specifies a list of nrf_datacube instances that hold the results produced in the run.

Formal propositions:

- has_unique_id: The id shall be unique in the exchange dataset.

4.2.8.6 TYPE nrf_quantity_sequencing_type

An **nrf_quantity_sequencing_type** specifies four mutually exclusive sequencing constraints for a quantity value list:
strictly decreasing values, that is $x_{i+1} < x_i$,
monotonic decreasing values, that is $x_{i+1} \leq x_i$,
monotonic increasing values, that is $x_{i+1} \geq x_i$,
strictly increasing values, that is $x_{i+1} > x_i$.

Express specification:


```
TYPE nrf_quantity_sequencing_type = ENUMERATION OF(
  STRICTLY_DECREASING,
  MONOTONIC_DECREASING,
  MONOTONIC_INCREASING,
  STRICTLY_INCREASING);
END_TYPE;
```

4.2.8.7 ENTITY nrf_state_list

An **nrf_state_list** specifies the quantity type and the quantity values that identify each of the known discrete states of a system that is represented by a collection of observable items.

EXAMPLE For an analysis run where results are stored at certain sample points in time, the state quantity_type would be an nrf_real_quantity_type for *time*, e.g. qualified as *mission_elapsed_time*, the quantity_sequencing would be STRICTLY_INCREASING and the real_values would contain the sample time values in the unit specified in the nrf_real_quantity_type for *mission_elapsed_time*.

Express specification:

```
ENTITY nrf_state_list;
  quantity_type       : nrf_any_quantity_type;
  quantity_sequencing : OPTIONAL nrf_quantity_sequencing_type;
  real_values         : LIST OF REAL;
  integer_values      : LIST OF INTEGER;
DERIVE
  number_of_states    : INTEGER := nrf_derive_number_of_states_in_state_list(SELF);
WHERE
  has_correct_number_of_real_values:
    SIZEOF(real_values) = number_of_states * quantity_type.number_of_real_values;
  has_correct_number_of_integer_values:
    SIZEOF(integer_values) = number_of_states * quantity_type.number_of_integer_values;
  has_valid_quantity_type_for_sequencing_type:
    (NOT EXISTS(quantity_sequencing)) OR
    (EXISTS(quantity_sequencing) AND
      ('NRF_ARM.NRF_PHYSICAL_QUANTITY_TYPE' IN TYPEOF(quantity_type)));
  has_valid_state_value_sequencing:
    nrf_verify_state_value_sequencing(SELF);
END_ENTITY;
```

Attribute definitions:

- quantity_type specifies the quantity type used to identify the state of the system.
- quantity_sequencing optionally specifies a sequencing constraint for the state values.
- real_values specifies for each identified state the value(s) for the real valued elements of the quantity_type.
- integer_values specifies for each identified state the value(s) for the integer valued elements of the quantity_type.

Formal propositions:

- has_correct_number_of_real_values: The number of real_values shall be the product of the number of states times and the required number_of_real_values of the quantity_type.
- has_correct_number_of_integer_values: The number of integer_values shall be the product of the number of states times and the required number_of_integer_values of the quantity_type.

— has_valid_quantity_type_for_sequencing_type: The state quantity_type shall be an nrf_physical_quantity_type when quantity_sequencing is specified.

— has_valid_state_value_sequencing: The sequencing of the state values shall conform to quantity_sequencing, if that is specified.

4.2.8.8 TYPE nrf_datacube_order_type

An **nrf_datacube_order_type** specifies the order in which indices for each of the bases of an nrf_datacube into the values attribute of an nrf_datacube are applied. The slowest changing index is with the first basis, the fastest changing index is with the last basis.

Express specification:

```
TYPE nrf_datacube_order_type = ENUMERATION OF(
  STATES_ITEMS_QUANTITIES,
  STATES_QUANTITIES_ITEMS,
  ITEMS_STATES_QUANTITIES,
  QUANTITIES_STATES_ITEMS,
  ITEMS_QUANTITIES_STATES,
  QUANTITIES_ITEMS_STATES);
END_TYPE;
```

4.2.8.9 ENTITY nrf_datacube

An **nrf_datacube** specifies a structured collection of discrete quantity values. A datacube is a three dimensional mathematical space for storing a potentially large amount of properties, where a property is defined as a quantity value for a given quantity type for a given observable item in a given (discrete) state. The three bases of the datacube are:

— a basis of quantity types;

— a basis of observable items;

— a basis of states.

Property positions in the datacube are identified by a tuple of indices, one index for each basis. If an actual quantity type in the quantity basis is an nrf_any_tensor_quantity_type, then such a property position will comprise as many values as there are elements in the nrf_any_tensor_quantity_type. The actual quantity values are stored in a flat list of values. The ordering in this flat list is determined by the value_order attribute. In order to cater for efficient storage of results data in different modes, all permutations of ordering the three bases are supported, as defined in nrf_datacube_order_type.

NOTE The **nrf_datacube** structure is designed to be efficient for storing and accessing very large amounts of structured results data. It is also designed to be mappable to an efficient implementation in any programming language without any complicated adaptation. To accommodate the latter, the flat list of values is split into a real and an integer list, in stead of having one LIST OF NUMBER, because the EXPRESS concept of NUMBER cannot be mapped efficiently into typical programming languages.

Express specification:

```
ENTITY nrf_datacube;
  name             : nrf_label;
  security_class   : OPTIONAL nrf_security_classification_level;
  value_order      : nrf_datacube_order_type;
  quantity_basis   : nrf_quantity_type_list;
```



```
  item_basis        : nrf_observable_item_list;
  state_basis       : nrf_state_list;
  real_values       : LIST OF REAL;
  integer_values    : LIST OF INTEGER;
WHERE
  has_correct_number_of_real_values:
    SIZEOF(real_values) = quantity_basis.number_of_real_values
    * SIZEOF(item_basis.items) * state_basis.number_of_states;
  has_correct_number_of_integer_values:
    SIZEOF(integer_values) = quantity_basis.number_of_integer_values
    * SIZEOF(item_basis.items) * state_basis.number_of_states;
END_ENTITY;
```

Attribute definitions:

- name specifies the human-interpretable name of an instance of an **nrf_datacube**
- security_class optionally specifies an nrf_security_classification_level for an **nrf_datacube**
- value_order specifies the order of the values in an **nrf_datacube** subtype instance.
- quantity_basis specifies the list of quantity types that define the quantity type basis of an **nrf_datacube**.
- item_basis specifies the list of observable items that define the item basis of an **nrf_datacube**.
- state_basis specifies the state quantity type and the list of state values that define the state basis of an **nrf_datacube**.
- real_values specifies the literal values for each of the property positions in an **nrf_datacube** that has a corresponding REAL value.
- integer_values specifies the literal values for each of the property positions in an **nrf_datacube** that has a corresponding INTEGER value.

Formal propositions:

- has_correct_number_of_real_values: The number of real values shall be equal to the product of the number of real values in the quantity_basis, the number of items in the item_basis and the number of states in the state_basis.
- has_correct_number_of_integer_values: The number of integer values shall be equal to the product of the number of integer values in the quantity_basis, the number of items in the item_basis and the number of states in the state_basis.

4.2.8.10 ENTITY nrf_derivation_procedure

An **nrf_derivation_procedure** specifies a named procedure that is used to derive one nrf_datacube from another nrf_data_cube.

Express specification:

```
ENTITY nrf_derivation_procedure;
  name : nrf_label;
UNIQUE
  has_unique_name: name;
END_ENTITY;
```

Attribute definitions:

— name specifies the human-interpretable name of an instance of **nrf_derivation_procedure**.

Formal propositions:

— has_unique_name: The name shall be unique in the exchange dataset.

4.2.8.11 ENTITY nrf_datacube_derivation_relationship

An **nrf_datacube_derivation_relationship** specifies a relationship between two nrf_datacube instances where one nrf_datacube was derived from an original one through the application of an nrf_derivation_procedure.

Express specification:

```
ENTITY nrf_datacube_derivation_relationship;
  original             : nrf_datacube;
  derived              : nrf_datacube;
  derivation_procedure : nrf_derivation_procedure;
END_ENTITY;
```

Attribute definitions:

— original specifies an nrf_datacube which is the source of the derivation relationship.

— derived specifies an nrf_datacube which is the result of the derivation relationship.

— derivation_procedure specifies a used defined procedure that was used to perform the derivation.

4.2.8.12 ENTITY nrf_network_model_nodes_mapping

An **nrf_network_model_nodes_mapping** specifies a mapping relation between the nodes of two nrf_network_model instances: one being the source model and the other being the target model. Its typical intended use is to map quantities between two related models, for example from a detailed to a course analysis model of the same object, or, from an analysis model to a test article representation.

Express specification:

```
ENTITY nrf_network_model_nodes_mapping;
  id                     : nrf_identifier;
  name                   : nrf_label;
  description            : nrf_text;
  source_model           : nrf_network_model;
  target_model           : nrf_network_model;
  node_mappings          : LIST OF nrf_network_node_mapping;
  relative_weights       : BOOLEAN;
  intended_quantity_types : LIST OF nrf_any_quantity_type;
UNIQUE
  has_unique_id_for_source_model: source_model, id;
WHERE
  has_valid_source_and_target_nodes: nrf_verify_nodes_in_mapping(SELF);
END_ENTITY;
```

Attribute definitions:

— id specifies the identifier of an instance of nrf_network_model_mapping.

— name specifies the human-interpretable name of an instance of nrf_network_model_mapping.

- description specifies the textual description of an instance of nrf_network_model_mapping.
- source_model specifies the nrf_network_model that is the source for the mapping.
- target_model specifies the nrf_network_model that is the target for the mapping.
- node_mappings specifies the list of source to target node mappings including weighting factors.
- relative_weights specifies whether the weights in the node_mappings are relative or absolute. If relative_weights is set to TRUE then the actual weights to be applied for each node mapping shall be divided by the sum of all weights for a given node mapping. Otherwise if relative_weights is set to FALSE then all weights for a given node mapping shall be applied as is.
- intended_quantity_types specifies the list of quantity types for which this mapping is intended. This list may be empty.

Formal propositions:

- has_unique_id_for_source_model: The combination of source_model and id shall be unique in the exchange dataset.
- has_valid_source_and_target_nodes: All source and target nodes in node_mappings shall be part of the source_model and target_model respectively.

4.2.8.13 ENTITY nrf_network_node_mapping

An **nrf_network_node_mapping** specifies a mapping from one or more source nodes to a target node, including a list of weighting factors for all source nodes. An **nrf_network_node_mapping** is always part of an nrf_network_model_nodes_mapping.

Express specification:

```
ENTITY nrf_network_node_mapping;
  weights      : LIST [1:?] OF REAL;
  source_nodes : LIST [1:?] OF nrf_network_node;
  target_node  : nrf_network_node;
WHERE
  number_of_weights_and_source_nodes_match: SIZEOF(weights) = SIZEOF(source_nodes);
END_ENTITY;
```

Attribute definitions:

- weights specifies the list of weighting factors for each of the source_nodes.
- source_nodes specifies the list of one or more source nodes.
- target_nodes specifies the target node.

Formal propositions:

- number_of_weights_and_source_nodes_match: The number of weights shall be the same as the number of source_nodes.

4.2.8.14 RULE nrf_root_is_singleton

The RULE **nrf_root_is_singleton** verifies that the dataset contains only one single nrf_root instance.

Express specification:


```
RULE nrf_root_is_singleton FOR (nrf_root);
WHERE
  root_is_singleton: SIZEOF(nrf_root) = 1;
END_RULE;
```

4.2.8.15 FUNCTION nrf_verify_initializations

The function **nrf_verify_initializations** verifies that the nrf_network_node and nrf_network_node_relationship instances referenced in the initializations of a given nrf_case are all part of the submodel tree of the nrf_network_model associated with the nrf_case. If this is the case the function returns TRUE, otherwise it returns FALSE.

Express specification:

```
FUNCTION nrf_verify_initializations(a_case : nrf_case) : BOOLEAN;
  REPEAT i := 1 TO SIZEOF(a_case.initializations);
    REPEAT j := 1 TO SIZEOF(a_case.initializations[i].item_basis.items);
      IF NOT (nrf_verify_item_in_model_tree(
          a_case.initializations[i].item_basis.items[j],
          a_case.for_model)) THEN
        RETURN(FALSE);
      END_IF;
    END_REPEAT;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— a_case specifies the nrf_case for which the initializations shall be verified.

4.2.8.16 FUNCTION nrf_verify_unique_identifiers

The function **nrf_verify_unique_identifiers** verifies that the id attributes of the instances in a given list are unique. It is a generic function that can be used on any list of instances of an ENTITY that has an attribute id of TYPE STRING. If the id's are unique the function returns TRUE, otherwise it returns FALSE.

Express specification:

```
FUNCTION nrf_verify_unique_identifiers(
    a_list : LIST OF GENERIC) : BOOLEAN;
  LOCAL
    id_list : LIST OF STRING := [];
  END_LOCAL;
  REPEAT i := 1 TO SIZEOF(a_list);
    id_list := id_list + a_list[i].id;
  END_REPEAT;
  IF NOT VALUE_UNIQUE(id_list) THEN
    RETURN(FALSE);
  END_IF;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— a_list specifies the list of instances to be verified.

4.2.8.17 FUNCTION nrf_verify_item_in_model_tree

The function **nrf_verify_item_in_model_tree** verifies that an nrf_observable_item is a valid constituent of an nrf_network_model. It returns FALSE if the given nrf_observable_item is an nrf_network_node or an nrf_network_node_relationship and is not part of the submodel tree of the given nrf_network_model. Otherwise the function returns TRUE.

Express specification:

```
FUNCTION nrf_verify_item_in_model_tree(
    an_item : nrf_observable_item;
    a_model : nrf_network_model) : BOOLEAN;
  LOCAL
    valid : BOOLEAN;
  END_LOCAL;

  IF ('NRF_ARM.NRF_NETWORK_NODE' IN TYPEOF(an_item)) THEN
    IF NOT (an_item IN a_model.nodes) THEN
      valid := FALSE;
      REPEAT i := 1 TO SIZEOF(a_model.submodels) WHILE (NOT valid);
        IF nrf_verify_item_in_model_tree(an_item, a_model.submodels[i]) THEN
          valid := TRUE;
        END_IF;
      END_REPEAT;
      IF NOT valid THEN
        RETURN(FALSE);
      END_IF;
    END_IF;
  END_IF;

  IF ('NRF_ARM.NRF_NETWORK_NODE_RELATIONSHIP' IN TYPEOF(an_item)) THEN
    IF NOT (an_item IN a_model.node_relationships) THEN
      valid := FALSE;
      REPEAT i := 1 TO SIZEOF(a_model.submodels) WHILE (NOT valid);
        IF nrf_verify_item_in_model_tree(an_item, a_model.submodels[i]) THEN
          valid := TRUE;
        END_IF;
      END_REPEAT;
      IF NOT valid THEN
        RETURN(FALSE);
      END_IF;
    END_IF;
  END_IF;

  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- an_item specifies the nrf_observable_item instance to be validated
- a_model specifies the nrf_network_model that is the root of the submodel tree to which an_item should belong

4.2.8.18 FUNCTION nrf_verify_acyclic_case_tree

The function **nrf_verify_acyclic_case_tree** verifies that there is no circular reference in the submodel tree of an nrf_case. In other words, the submodel tree forms an acyclic graph. The FUNCTION returns TRUE if this is the case and FALSE otherwise.

Express specification:

```
FUNCTION nrf_verify_acyclic_case_tree(
    a_case : nrf_case;
    supercases : LIST [1:?] OF UNIQUE nrf_case) : BOOLEAN;

  REPEAT i := 1 TO SIZEOF(a_case.subcases);
    IF (a_case.subcases[i] IN supercases) THEN
      RETURN(FALSE);
    END_IF;
    IF NOT nrf_verify_acyclic_case_tree(
        a_case.subcases[i], supercases + a_case.subcases) THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- a_case specifies the candidate nrf_case that is to be verified.
- supercases specifies the list of nrf_case instances that occur at the same or higher level as a_case.

4.2.8.19 FUNCTION nrf_derive_number_of_states_in_state_list

The function **nrf_derive_number_of_states_in_state_list** computes and returns the number of states in an nrf_state_list.

Express specification:

```
FUNCTION nrf_derive_number_of_states_in_state_list(
    sl : nrf_state_list) : INTEGER;
  IF (sl.quantity_type.number_of_real_values > 0) THEN
    RETURN(SIZEOF(sl.real_values) / sl.quantity_type.number_of_real_values);
  END_IF;
  IF (sl.quantity_type.number_of_integer_values > 0) THEN
    RETURN(SIZEOF(sl.integer_values) / sl.quantity_type.number_of_integer_values);
  END_IF;
  RETURN(0);
END_FUNCTION;
```

Argument definitions:

- sl specifies the nrf_state_list for which the number of states shall be computed.

4.2.8.20 FUNCTION nrf_verify_state_value_sequencing

The function **nrf_verify_state_value_sequencing** verifies the correct ordering of the values of an nrf_state_list according to its quantity_sequencing, if applicable. It returns TRUE if correct and FALSE otherwise.

Express specification:

```
FUNCTION nrf_verify_state_value_sequencing(
    sl : nrf_state_list) : BOOLEAN;
  LOCAL
```



```
    stride : INTEGER;
  END_LOCAL;

  IF NOT EXISTS(sl.quantity_sequencing) THEN
    RETURN(TRUE);
  END_IF;

  IF ('NRF_ARM.NRF_REAL_QUANTITY_TYPE' IN TYPEOF(sl.quantity_type)) THEN
    stride := sl.quantity_type.number_of_real_values;
    CASE sl.quantity_sequencing OF
      nrf_quantity_sequencing_type.STRICTLY_DECREASING:
        REPEAT i := (1 + stride) TO SIZEOF(sl.real_values) BY stride;
          IF sl.real_values[i] >= sl.real_values[i-stride] THEN
            RETURN(FALSE);
          END_IF;
        END_REPEAT;

      nrf_quantity_sequencing_type.MONOTONIC_DECREASING:
        REPEAT i := (1 + stride) TO SIZEOF(sl.real_values) BY stride;
          IF sl.real_values[i] > sl.real_values[i-stride] THEN
            RETURN(FALSE);
          END_IF;
        END_REPEAT;

      nrf_quantity_sequencing_type.MONOTONIC_INCREASING:
        REPEAT i := (1 + stride) TO SIZEOF(sl.real_values) BY stride;
          IF sl.real_values[i] < sl.real_values[i-stride] THEN
            RETURN(FALSE);
          END_IF;
        END_REPEAT;

      nrf_quantity_sequencing_type.STRICTLY_INCREASING:
        REPEAT i := (1 + stride) TO SIZEOF(sl.real_values) BY stride;
          IF sl.real_values[i] <= sl.real_values[i-stride] THEN
            RETURN(FALSE);
          END_IF;
        END_REPEAT;
    END_CASE;

    RETURN(TRUE);
  END_IF;

  IF ('NRF_ARM.NRF_INTEGER_QUANTITY_TYPE' IN TYPEOF(sl.quantity_type)) THEN
    stride := sl.quantity_type.number_of_integer_values;
    CASE sl.quantity_sequencing OF
      nrf_quantity_sequencing_type.STRICTLY_DECREASING:
        REPEAT i := (1 + stride) TO SIZEOF(sl.integer_values) BY stride;
          IF sl.integer_values[i] >= sl.integer_values[i-stride] THEN
            RETURN(FALSE);
          END_IF;
        END_REPEAT;

      nrf_quantity_sequencing_type.MONOTONIC_DECREASING:
        REPEAT i := (1 + stride) TO SIZEOF(sl.integer_values) BY stride;
          IF sl.integer_values[i] > sl.integer_values[i-stride] THEN
            RETURN(FALSE);
          END_IF;
        END_REPEAT;

      nrf_quantity_sequencing_type.MONOTONIC_INCREASING:
        REPEAT i := (1 + stride) TO SIZEOF(sl.integer_values) BY stride;
          IF sl.integer_values[i] < sl.integer_values[i-stride] THEN
            RETURN(FALSE);
          END_IF;
```

```
        END_REPEAT;

      nrf_quantity_sequencing_type.STRICTLY_INCREASING:
        REPEAT i := (1 + stride) TO SIZEOF(sl.integer_values) BY stride;
          IF sl.integer_values[i] <= sl.integer_values[i-stride] THEN
            RETURN(FALSE);
          END_IF;
        END_REPEAT;
    END_CASE;

    RETURN(TRUE);
  END_IF;

  RETURN(FALSE);

END_FUNCTION;
```

Argument definitions:

— sl specifies the candidate nrf_state_list to be verified.

4.2.8.21 RULE nrf_valid_values_in_datacubes

The RULE **nrf_valid_values_in_datacubes** verifies that all nrf_datacube instances contain valid quantity values.

Express specification:

```
RULE nrf_valid_values_in_datacubes FOR (nrf_root, nrf_datacube);
LOCAL
  rule_satisfied : LOGICAL := TRUE;
END_LOCAL;
  REPEAT i := 1 TO SIZEOF(nrf_datacube) WHILE rule_satisfied;
    IF NOT nrf_verify_values_in_datacube(nrf_root[1], nrf_datacube[i]) THEN
      rule_satisfied := FALSE;
    END_IF;
  END_REPEAT;
WHERE
  datacubes_have_valid_values: rule_satisfied;
END_RULE;
```

4.2.8.22 FUNCTION nrf_verify_values_in_datacube

The function **nrf_verify_values_in_datacube** verifies that all values in a given datacube are valid for the quantity type corresponding to the position in the real_values or integer_values list. The function returns TRUE if this is the case, otherwise it returns FALSE.

NOTE The functions **nrf_verify_values_in_datacube** and nrf_verify_values_for_quantity_type also provide examples for the implementation of algorithms to use an nrf_datacube.

Express specification:

```
FUNCTION nrf_verify_values_in_datacube(
  root : nrf_root;
  dc   : nrf_datacube) : BOOLEAN;

  LOCAL
    n_states       : INTEGER;
```



```
    n_items        : INTEGER;
    n_quantities   : INTEGER;
    real_offset    : INTEGER := 0;
    integer_offset : INTEGER := 0;
    qt             : nrf_any_quantity_type;
  END_LOCAL;

  n_states := SIZEOF(dc.state_basis.real_values);
  n_items := SIZEOF(dc.item_basis.items);
  n_quantities := SIZEOF(dc.quantity_basis.quantity_types);

  CASE dc.value_order OF

    nrf_datacube_order_type.STATES_ITEMS_QUANTITIES:
      REPEAT i_s := 1 TO n_states;
        REPEAT i_i := 1 TO n_items;
          REPEAT i_q := 1 TO n_quantities;
            qt := dc.quantity_basis.quantity_types[i_q];
            IF NOT nrf_verify_values_for_quantity_type(
                root, dc, real_offset, integer_offset, qt) THEN
              RETURN(FALSE);
            END_IF;
            real_offset := real_offset + qt.number_of_real_values;
            integer_offset := integer_offset + qt.number_of_integer_values;
          END_REPEAT;
        END_REPEAT;
      END_REPEAT;

    nrf_datacube_order_type.STATES_QUANTITIES_ITEMS:
      REPEAT i_s := 1 TO n_states;
        REPEAT i_q := 1 TO n_quantities;
          qt := dc.quantity_basis.quantity_types[i_q];
          REPEAT i_i := 1 TO n_items;
            IF NOT nrf_verify_values_for_quantity_type(
                root, dc, real_offset, integer_offset, qt) THEN
              RETURN(FALSE);
            END_IF;
            real_offset := real_offset + qt.number_of_real_values;
            integer_offset := integer_offset + qt.number_of_integer_values;
          END_REPEAT;
        END_REPEAT;
      END_REPEAT;

    nrf_datacube_order_type.ITEMS_STATES_QUANTITIES:
      REPEAT i_i := 1 TO n_items;
        REPEAT i_s := 1 TO n_states;
          REPEAT i_q := 1 TO n_quantities;
            qt := dc.quantity_basis.quantity_types[i_q];
            IF NOT nrf_verify_values_for_quantity_type(
                root, dc, real_offset, integer_offset, qt) THEN
              RETURN(FALSE);
            END_IF;
            real_offset := real_offset + qt.number_of_real_values;
            integer_offset := integer_offset + qt.number_of_integer_values;
          END_REPEAT;
        END_REPEAT;
      END_REPEAT;

    nrf_datacube_order_type.QUANTITIES_STATES_ITEMS:
      REPEAT i_q := 1 TO n_quantities;
        qt := dc.quantity_basis.quantity_types[i_q];
        REPEAT i_s := 1 TO n_states;
          REPEAT i_i := 1 TO n_items;
            IF NOT nrf_verify_values_for_quantity_type(
```

```
                root, dc, real_offset, integer_offset, qt) THEN
              RETURN(FALSE);
            END_IF;
            real_offset := real_offset + qt.number_of_real_values;
            integer_offset := integer_offset + qt.number_of_integer_values;
          END_REPEAT;
        END_REPEAT;
      END_REPEAT;

    nrf_datacube_order_type.ITEMS_QUANTITIES_STATES:
      REPEAT i_i := 1 TO n_items;
        REPEAT i_q := 1 TO n_quantities;
          qt := dc.quantity_basis.quantity_types[i_q];
          REPEAT i_s := 1 TO n_states;
            IF NOT nrf_verify_values_for_quantity_type(
                root, dc, real_offset, integer_offset, qt) THEN
              RETURN(FALSE);
            END_IF;
            real_offset := real_offset + qt.number_of_real_values;
            integer_offset := integer_offset + qt.number_of_integer_values;
          END_REPEAT;
        END_REPEAT;
      END_REPEAT;

    nrf_datacube_order_type.QUANTITIES_ITEMS_STATES:
      REPEAT i_q := 1 TO n_quantities;
        qt := dc.quantity_basis.quantity_types[i_q];
        REPEAT i_i := 1 TO n_items;
          REPEAT i_s := 1 TO n_states;
            IF NOT nrf_verify_values_for_quantity_type(
                root, dc, real_offset, integer_offset, qt) THEN
              RETURN(FALSE);
            END_IF;
            real_offset := real_offset + qt.number_of_real_values;
            integer_offset := integer_offset + qt.number_of_integer_values;
          END_REPEAT;
        END_REPEAT;
      END_REPEAT;

  END_CASE;

  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- root specifies the nrf_root instance.
- dc specifies the candidate nrf_datacube for which to verify the values.

4.2.8.23 FUNCTION nrf_verify_values_for_quantity_type

The function **nrf_verify_values_for_quantity_type** verifies that the values for a quantity_type in an nrf_datacube are valid. It is verified that values are of the correct type, and, if applicable, within the specified upper and lower bounds. When the actual parameter qt is an nrf_any_tensor_quantity_type, the function is called recursively for each of the tensor's elements.

Express specification:

```
FUNCTION nrf_verify_values_for_quantity_type(
    root           : nrf_root;
```

```
  dc              : nrf_datacube;
  real_offset     : INTEGER;
  integer_offset  : INTEGER;
  qt              : nrf_any_quantity_type) : BOOLEAN;

LOCAL
  typeof_qt           : SET OF STRING;
  rval                : REAL;
  rval2               : REAL;
  ival                : INTEGER;
  quantity_type_code  : INTEGER;
  j                   : INTEGER;
END_LOCAL;

typeof_qt := TYPEOF(qt);

IF ('NRF_ARM.NRF_REAL_QUANTITY_TYPE' IN typeof_qt) THEN
  quantity_type_code := 1;
ELSE
  IF ('NRF_ARM.NRF_INTEGER_QUANTITY_TYPE' IN typeof_qt) THEN
    quantity_type_code := 2;
  ELSE
    IF ('NRF_ARM.NRF_STRING_QUANTITY_TYPE' IN typeof_qt) THEN
      quantity_type_code := 3;
    ELSE
      IF ('NRF_ARM.NRF_ENUMERATION_QUANTITY_TYPE' IN typeof_qt) THEN
        quantity_type_code := 4;
      ELSE
        IF ('NRF_ARM.NRF_ANY_TENSOR_QUANTITY_TYPE' IN typeof_qt) THEN
          quantity_type_code := 5;
        ELSE
          RETURN(FALSE);
        END_IF;
      END_IF;
    END_IF;
  END_IF;
END_IF;

CASE quantity_type_code OF

1: -- nrf_real_quantity_type
BEGIN
  rval := dc.real_values[real_offset+1];
  IF (rval = root.undefined_real_sentinel) OR
      (rval = root.positive_infinity_real_sentinel) OR
      (rval = root.negative_infinity_real_sentinel) THEN
    RETURN(TRUE);
  END_IF;
  IF EXISTS(qt.lower_bound) THEN
    IF (rval < qt.lower_bound) THEN
      RETURN(FALSE);
    END_IF;
    IF (NOT(qt.lower_bound_inclusive)) AND
      (rval = qt.lower_bound) THEN
      RETURN(FALSE);
    END_IF;
  END_IF;
  IF EXISTS(qt.upper_bound) THEN
    IF (rval > qt.upper_bound) THEN
      RETURN(FALSE);
    END_IF;
    IF (NOT(qt.upper_bound_inclusive)) AND (rval = qt.upper_bound) THEN
      RETURN(FALSE);
    END_IF;
```

```
  END_IF;
END;

2: -- nrf_integer_quantity_type
BEGIN
  ival := dc.integer_values[integer_offset+1];
  IF (ival = root.undefined_integer_sentinel) OR
      (ival = root.positive_infinity_integer_sentinel) OR
      (ival = root.negative_infinity_integer_sentinel) THEN
    RETURN(TRUE);
  END_IF;
  IF EXISTS(qt.lower_bound) THEN
    IF (ival < qt.lower_bound) THEN
      RETURN(FALSE);
    END_IF;
    IF (NOT(qt.lower_bound_inclusive)) AND
      (ival = qt.lower_bound) THEN
      RETURN(FALSE);
    END_IF;
  END_IF;
  IF EXISTS(qt.upper_bound) THEN
    IF (ival > qt.upper_bound) THEN
      RETURN(FALSE);
    END_IF;
    IF (NOT(qt.upper_bound_inclusive)) AND
      (ival = qt.upper_bound)  THEN
      RETURN(FALSE);
    END_IF;
  END_IF;
END;

3: -- nrf_string_quantity_type
BEGIN
  ival := dc.integer_values[integer_offset+1];
  IF (ival < 1) OR (ival > SIZEOF(qt.string_values)) THEN
    RETURN(FALSE);
  END_IF;
END;

4: -- nrf_enumeration_quantity_type
BEGIN
  ival := dc.integer_values[integer_offset+1];
  IF (ival < 1) OR (ival > SIZEOF(qt.enumeration_items)) THEN
    RETURN(FALSE);
  END_IF;
END;

5: -- nrf_any_tensor_quantity_type
BEGIN
  -- recursively verify the quantity types and values contained in the tensor
  REPEAT i := 1 TO nrf_get_required_number_of_elements_in_any_tensor(qt);
    IF SIZEOF(qt.elements) = 1 THEN
      -- special case: tensor where all elements have the same quantity_type
      j := 1;
    ELSE
      j := i;
    END_IF;
    IF NOT(nrf_verify_values_for_quantity_type(
        root, dc, real_offset, integer_offset, qt.elements[j].quantity_type)) THEN
      RETURN(FALSE);
    END_IF;
    real_offset := real_offset +
      qt.elements[j].quantity_type.number_of_real_values;
    integer_offset := integer_offset +
```



```
        qt.elements[j].quantity_type.number_of_integer_values;
    END_REPEAT;
  END;

  END_CASE;

  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- root specifies the nrf_root instance.
- dc specifies the nrf_datacube for which a quantity_type and value shall be verified.
- real_offset specifies the offset of the actual quantity_type value into the real_values list of the datacube, i.e. the real_values[real_offset+1] is the value element to be verified.
- integer_offset specifies the offset of the actual quantity_type value into the integer_values list of the datacube, i.e. the integer_values[integer_offset+1] is the value element to be verified.
- qt specifies the quantity_type that defines the required type of the value(s) to be verified.

4.2.8.24 FUNCTION nrf_verify_nodes_in_mapping

The function **nrf_verify_nodes_in_mapping** verifies that all source and target nodes specified in the node_mappings of an nrf_network_model_nodes_mapping are part of the respective source_model and target_model. If this is the case the function returns TRUE, otherwise the function returns FALSE.

Express specification:

```
FUNCTION nrf_verify_nodes_in_mapping(
    a_mapping : nrf_network_model_nodes_mapping) : BOOLEAN;

  REPEAT i := 1 TO SIZEOF(a_mapping.node_mappings);
    REPEAT j := 1 TO SIZEOF(a_mapping.node_mappings[i].source_nodes);
      IF NOT nrf_verify_item_in_model_tree(
          a_mapping.node_mappings[i].source_nodes[j],
          a_mapping.source_model) THEN
        RETURN(FALSE);
      END_IF;
    END_REPEAT;
    IF NOT nrf_verify_item_in_model_tree(
        a_mapping.node_mappings[i].target_node,
        a_mapping.target_model) THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- a_mapping specifies the nrf_network_model_nodes_mapping to be verified.

4.2.9 NRF Product structure UoF

The UoF nrf_product_structure captures the structure of the product being modelled – e.g. the product assembly tree or a functional decomposition tree.

This Unit of Functionality is defined following as closely as possible the concepts for product and product structure definition in ISO 10303-41 and ISO 10303-44.

4.2.9.1 ENTITY nrf_product

An **nrf_product** is an identification and textual description of a physically realisable object that is produced by a natural or artificial process.

Express specification:

```
ENTITY nrf_product;
  id                : nrf_identifier;
  name              : nrf_label;
  description       : nrf_text;
  frame_of_reference : SET [1:?] OF nrf_product_context;
UNIQUE
  has_unique_id: id;
END_ENTITY;
```

Attribute definitions:

- id specifies the identifier of an instance of **nrf_product**.
- name specifies the human-interpretable name of an instance of **nrf_product**.
- description specifies the textual description of an instance of **nrf_product**.
- frame_of_reference specifies one or more references to a product_context.

Formal propositions:

- has_unique_id: The id shall be unique in the exchange dataset.

4.2.9.2 ENTITY nrf_product_context

An **nrf_product_context** collects information on the engineering or manufacturing perspective in which the product data are defined. This context may affect the meaning and usage of the product data.

Express specification:

```
ENTITY nrf_product_context;
  name            : nrf_label;
  discipline_type : nrf_label;
END_ENTITY;
```

Attribute definitions:

- name specifies the human-interpretable name of an instance of **nrf_product_context**.
- discipline_type specifies the identification of the discipline to which the product data belong. It is a label string.

4.2.9.3 ENTITY nrf_product_definition

An **nrf_product_definition** is an identification of a characterization of a product in a particular application context.

Express specification:

```
ENTITY nrf_product_definition;
  id                  : nrf_identifier;
  description         : nrf_text;
  frame_of_reference : nrf_product_definition_context;
UNIQUE
  has_unique_id_for_frame_of_reference: id, frame_of_reference;
END_ENTITY;
```

Attribute definitions:

- id specifies the identifier of an instance of **nrf_product_definition**.
- description specifies the textual description of an instance of **nrf_product_definition**.
- frame_of_reference specifies the reference to nrf_product_definition_context.

Formal propositions:

- has_unique_id_for_frame_of_reference: The combination of id and frame_of_reference shall be unique in the exchange dataset.

NOTE A product's physical design may be one product_definition while the functional design of the same product may be a different product_definition.

4.2.9.4 ENTITY nrf_product_definition_context

An **nrf_product_definition_context** collects information on the stage in the product life cycle to which the product data belongs. This context may affect the meaning and usage of the product data.

Express specification:

```
ENTITY nrf_product_definition_context;
  name             : nrf_label;
  life_cycle_stage : nrf_label;
END_ENTITY;
```

Attribute definitions:

- name specifies the human-interpretable name of an instance of **nrf_product_definition_context**.
- life_cycle_stage specifies a label string that identifies the life cycle stage to which the product data belong.

EXAMPLE Typical values for life_cycle_stage are: 'conceptual design', 'preliminary design', 'detailed design'.

4.2.9.5 ENTITY nrf_product_next_assembly_usage_relationship

An **nrf_product_next_assembly_usage_relationship** is an identified association between two product_definitions. It specifies that one nrf_product_definition is the next level higher assembly that uses the other nrf_product_definition as a constituent. This relationship is used to specify an assembly tree that is also often called product tree or manufacturing tree.

Express specification:


```
ENTITY nrf_product_next_assembly_usage_relationship;
  id          : nrf_identifier;
  assembly    : nrf_product_definition;
  constituent : nrf_product_definition;
UNIQUE
  has_unique_id_for_assembly: id, assembly;
WHERE
  product_assembly_is_acyclic:
    nrf_verify_acyclic_product_definition_relationship(SELF, [constituent]);
END_ENTITY;
```

Attribute definitions:

- id specifies the identifier of an instance of **nrf_product_next_assembly_usage_relationship**.
- assembly specifies an higher level nrf_product_definition.
- constituent specifies a lower level nrf_product_definition.

Formal propositions:

- has_unique_id_for_assembly: The combination id and assembly shall be unique in the exchange dataset.
- product_assembly_is_acyclic: The relationship shall form an acyclic graph, i.e. no circular references to nrf_product_definition instances are allowed.

4.2.9.6 FUNCTION nrf_verify_acyclic_product_definition_relationship

The function **nrf_verify_acyclic_product_definition_relationship** determines whether or not the given nrf_product_definition entities have been self-defined by the associations made in the specified nrf_product_next_assembly_usage_relationship.

The function returns a value of FALSE if an nrf_product_definition, which is referenced by the attribute assembly of the relation argument, is also referenced through the attribute constituent of the same relation argument or any of relation instances in the product_definition tree. Otherwise, it returns a value of TRUE.

Express specification:

```
FUNCTION nrf_verify_acyclic_product_definition_relationship(
    relation     : nrf_product_next_assembly_usage_relationship;
    constituents : SET [1:?] OF nrf_product_definition) : BOOLEAN;
  LOCAL
    x : BAG OF nrf_product_next_assembly_usage_relationship;
    s : STRING;
  END_LOCAL;
  IF relation.assembly IN constituents THEN
    RETURN(FALSE);
  END_IF;
  s := 'NRF_ARM.NRF_PRODUCT_NEXT_ASSEMBLY_USAGE_RELATIONSHIP.CONSTITUENT';
  x := USEDIN(relation.assembly, s);
  REPEAT i := 1 TO HIINDEX(x);
    IF NOT nrf_verify_acyclic_product_definition_relationship(
        x[i], constituents + [relation.constituent]) THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- relation: the candidate nrf_product_definition_relationship to be verified.
- constituents: the set of nrf_product_definition entity data types that the function is searching for in the constituent attribute of the relation argument.

4.2.9.7 ENTITY nrf_product_version

An **nrf_product_version** is an identified grouping of nrf_product_definitions that together define the version of a product. A version differs from other versions of a product in some significant way. However, the difference is insufficient to regard the version as a different product.

The rules for differentiating between the case in which the changes to a product are great enough to justify the definition of a new product and the case in which the changes only require the definition of a new version of a product are enterprise specific. Such rules are not specified in the scope of this protocol.

Express specification:

```
ENTITY nrf_product_version;
  for_product : nrf_product;
  id          : nrf_identifier;
  description : nrf_text;
  definitions : SET [1:?] OF nrf_product_definition;
UNIQUE
  has_unique_id_for_product: id, for_product;
END_ENTITY;
```

Attribute definitions:

- for_product specifies the reference to an nrf_product.
- id specifies the identifier of an instance of **nrf_product_version**.
- description specifies the textual description of an instance of **nrf_product_version**.
- definitions specifies the references to one or more nrf_product_definitions.

Formal propositions:

- has_unique_id_for_product: The combination of for_product and id shall be unique in the exchange dataset.

NOTE The current definition in [STEP-41] for product_definition allows referencing only one version of the product (product_definition_formation).

EXAMPLE The version identifier 'v3.11' for a part is an example of an id.

4.2.10 Materials UoF

The nrf_materials UoF contains the application objects needed to specify identified materials and specify the required physical properties to be associated with them.

4.2.10.1 ENTITY nrf_material_class

An **nrf_material_class** is a specification of a class of nrf_material instances, that is a named category of materials which share common characteristics and behaviour.

Express specification:

```
ENTITY nrf_material_class
  SUBTYPE OF(nrf_named_observable_item_class);
  subclasses                    : LIST OF nrf_material_class;
  required_quantity_type_names : LIST OF nrf_non_blank_label;
DERIVE
  all_required_quantity_type_names : LIST OF nrf_non_blank_label :=
    nrf_get_all_required_quantity_type_names(SELF);
WHERE
  material_class_tree_is_acyclic:
    nrf_verify_acyclic_material_class_tree(SELF, [SELF]);
  required_quantity_type_names_are_unique:
    VALUE_UNIQUE(required_quantity_type_names);
END_ENTITY;
```

Attribute definitions:

- subclasses specifies the subclasses of an **nrf_material_class**, if any. This allows for the definition of a class hierarchy for materials.
- required_quantity_type_names specifies the names of the quantity types (nrf_any_quantity_type instances) that shall be defined for materials that belong to an **nrf_material_class**.
- all_required_quantity_type_names is derived to return the expanded list of all names for the required quantity types for the current **nrf_material_class** and all its subclasses.

Formal propositions:

- material_class_tree_is_acyclic: The hierarchical material_class tree shall be acyclic, i.e. there shall be no circular references.
- required_quantity_type_names_are_unique: The required_quantity_type_names shall be unique.

4.2.10.2 ENTITY nrf_material

An **nrf_material** is a simple specification of a material. Materials may be defined for use in nrf_network_model instances and can be assigned to model elements. Material properties can be specified as quantity type values.

Express specification:

```
ENTITY nrf_material
  SUBTYPE OF(nrf_named_observable_item);
  SELF\nrf_named_observable_item.item_class : nrf_material_class;
END_ENTITY;
```

Attribute definitions:

- item_class specifies the nrf_network_material_class identifiying the class of an **nrf_material**

4.2.10.3 FUNCTION nrf_get_all_required_quantity_type_names

The function **nrf_get_all_required_quantity_type_names** collects and returns all required_quantity_type_names defined in an nrf_material_class and its subclasses. The collection is done recursively, starting from the top of the material class tree.

Express specification:

```
FUNCTION nrf_get_all_required_quantity_type_names(
    mc : nrf_material_class) : LIST OF nrf_non_blank_label;
  LOCAL
    all_required_quantity_type_names : LIST OF nrf_non_blank_label;
  END_LOCAL;
  all_required_quantity_type_names :=
      mc.required_quantity_type_names;
  REPEAT i := 1 TO SIZEOF(mc.subclasses);
    all_required_quantity_type_names :=
      all_required_quantity_type_names +
      nrf_get_all_required_quantity_type_names(mc.subclasses[i]);
  END_REPEAT;
  RETURN(all_required_quantity_type_names);
END_FUNCTION;
```

Argument definitions:

— mc specifies the nrf_material_class for which all required_quantity_type_names need to be returned.

4.2.10.4 FUNCTION nrf_verify_acyclic_material_class_tree

The function **nrf_verify_acyclic_material_class_tree** verifies recursively that the hierarchical class tree defined by nrf_material_class instances is acyclic, i.e. there are no circular nrf_material_class references. The function returns TRUE if this is the case and FALSE otherwise.

Express specification:

```
FUNCTION nrf_verify_acyclic_material_class_tree(
    mc : nrf_material_class;
    superclasses : LIST OF nrf_material_class) : BOOLEAN;
  REPEAT i := 1 TO SIZEOF(mc.subclasses);
    IF mc.subclasses[i] IN superclasses THEN
      RETURN(FALSE);
    END_IF;
    IF NOT nrf_verify_acyclic_material_class_tree(
       mc.subclasses[i], superclasses + mc.subclasses) THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— mc specifies the nrf_material_class to be verified.

— superclasses specifies the list of higher level classes that shall not be referenced by mc or its subclasses.

4.2.11 END_SCHEMA declaration for nrf_arm

The following EXPRESS declaration ends the nrf_arm schema.

Express specification:

```
END_SCHEMA;
```

4.3 Meshed geometric model (MGM) module

This subclause specifies the units of functionality for the meshed geometric model module. The module contains the following units of functionality:

1. MGM visual presentation
2. MGM basic geometry objects
3. MGM meshed geometric model
4. MGM meshed boolean construction geometry

4.3.1 SCHEMA declaration for mgm_arm

The following EXPRESS declaration begins the mgm_arm schema.

Express specification:

```
SCHEMA mgm_arm;
-- $Id$
-- Copyright (c) 1995-2018 European Space Agency (ESA)
-- All rights reserved.
```

4.3.2 Interfaced schema(ta) for mgm_arm

The mgm_arm schema uses the nrf_arm schema specified in [STEP-NRF].

Express specification:

```
USE FROM nrf_arm;
```

4.3.3 CONSTANT specifications

One constant is defined for the MGM module: SCHEMA_OBJECT_IDENTIFIER.

Express specification:

```
CONSTANT
  SCHEMA_OBJECT_IDENTIFIER2 : STRING :=
 '{http://www.purl.org/ESA/step-tas/v6.0/mgm_arm.exp}';
  -- in formal version to be replaced with
  -- '{ iso standard n part(p) version(v) }
END_CONSTANT;
```

Constant definitions:

SCHEMA_OBJECT_IDENTIFIER provides a built-in way to reference the object identifier of the protocol for version verification. For the definition and usage of the object identifier see ISO 10303-1 and Annex E.

4.3.4 MGM visual presentation UoF

The MGM visual presentation UoF collects the application objects that capture visualization aspects for use in the various model representations.

4.3.4.1 TYPE mgm_rgb_component

An **mgm_rgb_component** specifies a type for the value of the red, green or blue component of an mgm_colour_rgb.

Express specification:

```
TYPE mgm_rgb_component = REAL;
WHERE
  {0.0 <= SELF <= 1.0};
END_TYPE;
```

Formal propositions:

- wr1: The value of an **mgm_rgb_component** shall be between zero and one inclusive.

4.3.4.2 ENTITY mgm_colour_rgb

An **mgm_colour_rgb** defines a colour by specifying the intensity of red, green and blue.

Express specification:

```
ENTITY mgm_colour_rgb;
  name  : OPTIONAL nrf_label;
  red   : mgm_rgb_component;
  green : mgm_rgb_component;
  blue  : mgm_rgb_component;
END_ENTITY;
```

Attribute definitions:

- name optionally specifies the name of the colour.
- red specifies the intensity of the red component of the colour as a value between 0.0 and 1.0.
- green specifies the intensity of the green component of the colour as a value between 0.0 and 1.0.
- blue specifies the intensity of the blue component of the colour as a value between 0.0 and 1.0.

4.3.5 MGM basic geometry objects UoF

The MGM basic geometry objects UoF contains application objects providing basic entitites for the UoF mgm_meshed_geometric_model.

4.3.5.1 ENTITY mgm_3d_cartesian_point

An **mgm_3d_cartesian_point** specifies a point in a three-dimensional, Cartesian, orthonormal, right-handed coordinate system with x, y and z coordinates. An **mgm_3d_cartesian_point** may optionally have an identifier. An **mgm_3d_cartesian_point** can be referenced at any place in a shape definition where a point is demanded. The unit of length is specified through a given quantity_type, which is typically enforced to be the 'length' quantity type in an mgm_quantity_context.

Express specification:

```
ENTITY mgm_3d_cartesian_point
  SUPERTYPE OF(ONEOF(mgm_parametric_3d_cartesian_point));
  id            : OPTIONAL nrf_identifier;
  x             : REAL;
  y             : REAL;
  z             : REAL;
  quantity_type : nrf_real_quantity_type;
WHERE
  wr1: nrf_verify_dimensional_exponents(
    quantity_type.quantity_category,
    1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
END_ENTITY;
```

Attribute definitions:

- id optionally specifies the identifier of an instance of **mgm_3d_cartesian_point**.
- x specifies the value of the coordinate on the X axis.
- y specifies the value of the coordinate on the Y axis.
- z specifies the value of the coordinate on the Z axis.
- quantity_type specifies the quantity type of the x, y and z coordinates.

Formal propositions:

- wr1: The quantity_type shall have a quantity_category with dimension 'length'.

4.3.5.2 ENTITY mgm_parametric_3d_cartesian_point

An **mgm_parametric_3d_cartesian_point** is a type of mgm_3d_cartesian_point that specifies its coordinates with value prescriptions. This enables parametric definition of points.

Express specification:

```
ENTITY mgm_parametric_3d_cartesian_point
  SUBTYPE OF(mgm_3d_cartesian_point);
  x_prescription : nrf_real_quantity_value_prescription;
  y_prescription : nrf_real_quantity_value_prescription;
  z_prescription : nrf_real_quantity_value_prescription;
DERIVE
  SELF\mgm_3d_cartesian_point.x : REAL := x_prescription.val;
  SELF\mgm_3d_cartesian_point.y : REAL := y_prescription.val;
  SELF\mgm_3d_cartesian_point.z : REAL := z_prescription.val;
WHERE
  wr1: (x_prescription.quantity_type :=: quantity_type) AND
       (y_prescription.quantity_type :=: quantity_type) AND
```

```
      (z_prescription.quantity_type :=: quantity_type);
END_ENTITY;
```

Attribute definitions:

- x_prescription prescribes the value for the coordinate on the X axis
- y_prescription prescribes the value for the coordinate on the Y axis
- z_prescription prescribes the value for the coordinate on the Z axis
- x is derived from the literal or default value of x_prescription
- y is derived from the literal or default value of y_prescription
- z is derived from the literal or default value of z_prescription

Formal propositions:

- wr1: Each of the coordinate prescriptions shall have the same quantity_type as the quantity_type of the **mgm_parametric_3d_cartesian_point** itself.

4.3.5.3 ENTITY mgm_3d_direction

An **mgm_3d_direction** defines a general direction vector in three- dimensional Cartesian space. The actual magnitudes of the components have no effect upon the direction being defined, only the ratios x:y:z are significant.

NOTE The components of this entity are not necessarily normalised. If a unit vector is required it should be normalised before use.

Express specification:

```
ENTITY mgm_3d_direction
  SUPERTYPE OF(ONEOF(mgm_parametric_3d_direction));
  x : REAL;
  y : REAL;
  z : REAL;
WHERE
  wr1: SQRT(x**2 + y**2 + z**2) > 0.9;
END_ENTITY;
```

Attribute definitions:

- x specifies the component in the direction of the X axis.
- y specifies the component in the direction of the Y axis.
- z specifies the component in the direction of the Z axis.

Formal propositions:

- wr1: The magnitude of the direction vector shall be greater than 0.9.

NOTE A significant value for the magnitude of direction vector is required to prevent numerical problems. A minimum magnitude of 0.9 is a pragmatic choice that allows any normalisation method to produce a valid direction vector.

4.3.5.4 ENTITY mgm_parametric_3d_direction

An **mgm_parametric_3d_direction** specifies a general direction vector in three-dimensional Cartesian space with value prescriptions for the direction components. The actual magnitudes of the components have no effect upon the direction being defined, only the ratios x:y:z are significant.

NOTE The components of this entity are not necessarily normalised. If a unit vector is required it should be normalised before use.

Express specification:

```
ENTITY mgm_parametric_3d_direction
  SUBTYPE OF(mgm_3d_direction);
  x_prescription : nrf_real_quantity_value_prescription;
  y_prescription : nrf_real_quantity_value_prescription;
  z_prescription : nrf_real_quantity_value_prescription;
DERIVE
  SELF\mgm_3d_direction.x : REAL := x_prescription.val;
  SELF\mgm_3d_direction.y : REAL := y_prescription.val;
  SELF\mgm_3d_direction.z : REAL := z_prescription.val;
WHERE
  wr1: nrf_verify_dimensional_exponents(
 x_prescription.quantity_type.quantity_category,
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) AND
    (y_prescription.quantity_type :=: x_prescription.quantity_type) AND
    (z_prescription.quantity_type :=: x_prescription.quantity_type);
END_ENTITY;
```

Attribute definitions:

- x_prescription prescribes the value for the component in the direction of the X axis
- y_prescription prescribes the value for the component in the direction of the Y axis
- z_prescription prescribes the value for the component in the direction of the Z axis
- x is derived from the literal or default value of x_prescription
- y is derived from the literal or default value of y_prescription
- z is derived from the literal or default value of z_prescription

Formal propositions:

- wr1: All direction prescriptions shall have the same quantity_type and they shall all be non-dimensional.

4.3.5.5 ENTITY mgm_axis_transformation

An **mgm_axis_transformation** is an abstract entity that provides a generic mechanism to reference static coordinate system transformations between related geometrical shapes. Each **mgm_axis_transformation** is either an mgm_axis_placement or an mgm_axis_transformation_sequence.

Express specification:

```
ENTITY mgm_axis_transformation
  ABSTRACT SUPERTYPE OF(ONEOF(mgm_axis_placement, mgm_axis_transformation_sequence));
END_ENTITY;
```


4.3.5.6 ENTITY mgm_axis_placement

An **mgm_axis_placement** defines the location and orientation of a Cartesian orthonormal coordinate system in three-dimensional space with respect to the coordinate system in which it is specified.

From the attributes of an **mgm_axis_placement** a rotation matrix and a translation vector can be derived. In the following the attribute names are replaced by symbols in the following ways:

$\mathbf{t}$: location

$\mathbf{z}$: z_axis_direction

$\hat{\mathbf{x}}$: x_axis_direction

The vectors $\mathbf{x}$ and $\mathbf{y}$ are derived from the vectors $\hat{\mathbf{x}}$ and $\mathbf{z}$ via their cross-products:

$$\mathbf{y} = \mathbf{z} \times \hat{\mathbf{x}}$$

$$\mathbf{x} = \mathbf{y} \times \mathbf{z}$$

For the three vectors $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ we introduce now the normalized versions:

$$\widetilde{\mathbf{x}} = \frac{\mathbf{x}}{|\mathbf{x}|}, \widetilde{\mathbf{y}} = \frac{\mathbf{y}}{|\mathbf{y}|}, \widetilde{\mathbf{z}} = \frac{\mathbf{z}}{|\mathbf{z}|},$$

The rotation matrix to transform coordinates from the local to the reference system is built from these normalized vectors

$$\mathbf{R} = \begin{bmatrix} \widetilde{\mathbf{x}} & \widetilde{\mathbf{y}} & \widetilde{\mathbf{z}} \end{bmatrix}$$

Suppose $\mathbf{p}$ is a vector with coordinates expressed in the reference system and $\hat{\mathbf{p}}$ is the same vector, but its coordinates are expressed in the local coordinate system. The following relation between $\mathbf{p}$ and $\hat{\mathbf{p}}$ holds:

$$\mathbf{p} = \mathbf{R}\hat{\mathbf{p}} + \mathbf{t}$$

Express specification:

```
ENTITY mgm_axis_placement
  SUBTYPE OF(mgm_axis_transformation);
  location         : mgm_3d_cartesian_point;
  z_axis_direction : mgm_3d_direction;
  x_axis_direction : mgm_3d_direction;
WHERE
  wr1: ABS((z_axis_direction.x*x_axis_direction.x +
 z_axis_direction.y*x_axis_direction.y +
 z_axis_direction.z*x_axis_direction.z)/
 (SQRT(z_axis_direction.x**2 + z_axis_direction.y**2 + z_axis_direction.z**2)*
 SQRT(x_axis_direction.x**2 + x_axis_direction.y**2 + x_axis_direction.z**2)))
           < 0.9;
END_ENTITY;
```

Attribute definitions:

- location specifies a geometric_construction_point that serves as the origin of the coordinate system. The coordinates of location are expressed in the reference system.
- z_axis_direction specifies the exact direction of the z-axis of the local coordinate system to defined by an **mgm_axis_placement**. The components of z_axis_direction are expressed in the reference system.
- x_axis_direction specifies the (approximate) direction of the x-axis of the local coordinate system. x_axis_direction defines together with z_axis_direction the x-z-plane of the local coordinate system to be defined by an **mgm_axis_placement**. The components of x_axis_direction are expressed in the reference system.

Formal propositions:

- wr1: The absolute value of the dot product of the normalised versions of the z_axis_direction and the x_axis_direction shall be less than 0.9 in order to guarantee that the two direction vectors are not parallel or anti-parallel.

NOTE A value of 0.9 for the absolute value of the dot product of the normalised versions of the z_axis_direction and the x_axis_direction has been chosen to provide a significant deviation of 1.0, which corresponds to the parallel case. This allows practically any vector in the required x-z-plane that is not parallel to z_axis_direction to serve as x_axis_direction

4.3.5.7 ENTITY mgm_axis_transformation_sequence

An **mgm_axis_transformation_sequence** is a sequence of transformations of type mgm_translation_or_rotation, which can be either a translation or a rotation. An mgm_translation represents the translation. Two types of rotation can be part of the sequence: mgm_rotation_with_axes_fixed and mgm_rotation_with_axes_moving.

From each mgm_translation a translation vector $\hat{\mathbf{t}}$ can be created.

$$\hat{\mathbf{t}} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The occurence of an mgm_translation is added to translation vector build up from the previously defined mgm_translation instances in the sequence. Suppose the latter vector is referenced through $\widetilde{\mathbf{t}}$ the expression for the new translation vector $\mathbf{t}$ becomes:

$$\mathbf{t} = \widetilde{\mathbf{t}} + \hat{\mathbf{t}}$$

Both rotation types inherit the attributes from the super type mgm_rotation.

It is assumed that the normalized version of axis has the three components q_x , q_y and q_z , and the angle is represented by α. If the rotation is of type mgm_rotation_with_axes_fixed, rotation vector $\mathbf{r}$ is created from the α and $\mathbf{q}$:

$$\mathbf{r} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} = \begin{bmatrix} \alpha q_x \\ \alpha q_y \\ \alpha q_z \end{bmatrix}$$

If the rotation is of type mgm_rotation_with_axes_moving and the rotation is the *j-th* rotation in the **mgm_axis_transformation_sequence**, rotation vector r is then defined by:

$$\mathbf{r} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} = \alpha \mathbf{R}^{j-1}\mathbf{R}^{j-2}\ldots\mathbf{R}^{1} \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix}$$

with $\mathbf{R}^{j-1}$, $\mathbf{R}^{j-2}$, …, $\mathbf{R}^{1}$ being the rotation matrices of the previous rotations in the **mgm_axis_transformation_sequence**.

Suppose that the rotation vector $\mathbf{r}$ can be written as the product of its magnitude and a normalised version of the $\mathbf{r}$ referenced through $\mathbf{n}$:

$$\mathbf{r} = |\mathbf{r}|\mathbf{n} = |\mathbf{r}| \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix}$$

The *j-th* rotation matrix can be created with:

$$\mathbf{R}^j = \cos(|\mathbf{r}|)\mathbf{I} + [1 - \cos(|\mathbf{r}|)]\mathbf{n}\mathbf{n}^T + \sin(|\mathbf{r}|)\widetilde{\mathbf{r}}$$

$$\mathbf{R}^j = \cos(|\mathbf{r}|) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + [1-\cos(|\mathbf{r}|)] \begin{bmatrix} n_x n_x & n_y n_x & n_z n_x \\ n_x n_y & n_y n_y & n_z n_y \\ n_x n_z & n_y n_z & n_z n_x \end{bmatrix} + \sin(|\mathbf{r}|) \begin{bmatrix} 0 & -n_z & n_y \\ n_z & 0 & -n_x \\ -n_y & n_x & 0 \end{bmatrix}$$

The **mgm_axis_transformation_sequence** is a series of transformations, which must be executed in the order in which these occur in the sequence. If any mgm_translation instances occured in the sequence before any mgm_rotation, then the build up translation vector is also rotated using the rotation matrix of this mgm_rotation.

$$\mathbf{t} = \mathbf{R}^j\widetilde{\mathbf{t}}$$

In the following the position of a point expressed in the reference is indicated with $\mathbf{p}$ and the position of the same point expressed in the local system is indicated with $\hat{\mathbf{p}}$. For each of the transformations in **mgm_axis_transformation_sequence** the following operations have to be applied:

If the transformation is a rotation:

$$\mathbf{p} = \mathbf{R}^j\hat{\mathbf{p}}$$

If it is a translation:

$$\mathbf{p} = \hat{\mathbf{p}} + \mathbf{t}$$

in which $\hat{\mathbf{p}}$ becomes $\mathbf{p}$ of the previous transformation.

Express specification:

```
ENTITY mgm_axis_transformation_sequence
  SUBTYPE OF(mgm_axis_transformation);
  transformation_sequence : LIST [1:?] OF mgm_translation_or_rotation;
```



```
WHERE
  wr1: (SIZEOF(QUERY(ts <* transformation_sequence |
 'MGM_ARM.MGM_ROTATION_WITH_AXES_MOVING' IN TYPEOF(ts))) = 0) OR
       (SIZEOF(QUERY(ts <* transformation_sequence |
 'MGM_ARM.MGM_ROTATION_WITH_AXES_FIXED' IN TYPEOF(ts))) = 0);
END_ENTITY;
```

Attribute definitions:

- transformation_sequence specifies a sequence of coordinate transformation definitions of the type mgm_translation_or_rotation

Formal propositions:

- wr1: all mgm_rotation instances in transformation_sequence must be of the same subtype, i.e. either all mgm_rotation_with_axes_fixed or all mgm_rotation_with_axes_moving

4.3.5.8 ENTITY mgm_translation_or_rotation

An **mgm_translation_or_rotation** specifies an abstract supertype that provides a generic mechanism to reference an mgm_translation or an mgm_rotation.

Express specification:

```
ENTITY mgm_translation_or_rotation
  ABSTRACT SUPERTYPE OF(ONEOF(mgm_translation, mgm_rotation));
END_ENTITY;
```

4.3.5.9 ENTITY mgm_translation

An **mgm_translation** specifies a translation in three dimensional cartesian space expressed in x, y and z components.

NOTE An **mgm_translation** is only used in an mgm_axis_transformation_sequence.

Express specification:

```
ENTITY mgm_translation
  SUPERTYPE OF(ONEOF(mgm_parametric_translation))
  SUBTYPE OF(mgm_translation_or_rotation);
  x             : REAL;
  y             : REAL;
  z             : REAL;
  quantity_type : nrf_real_quantity_type;
WHERE
  wr1: nrf_verify_dimensional_exponents(
    quantity_type.quantity_category,
    1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
END_ENTITY;
```

Attribute definitions:

- x specifies translation in the x-direction of the reference system
- y specifies translation in the y-direction of the reference system
- z specifies translation in the z-direction of the reference system

— quantity_type specifies the quantity type of the x, y and z components.

Formal propositions:

— wr1: The quantity_type shall have a quantity_category with dimension 'length'.

4.3.5.10 ENTITY mgm_parametric_translation

An **mgm_parametric_translation** is a type of mgm_translation that specifies its components with value prescriptions. This enables parametric definition of translations.

Express specification:

```
ENTITY mgm_parametric_translation
  SUBTYPE OF(mgm_translation);
  x_prescription : nrf_real_quantity_value_prescription;
  y_prescription : nrf_real_quantity_value_prescription;
  z_prescription : nrf_real_quantity_value_prescription;
DERIVE
  SELF\mgm_translation.x : REAL := x_prescription.val;
  SELF\mgm_translation.y : REAL := y_prescription.val;
  SELF\mgm_translation.z : REAL := z_prescription.val;
WHERE
  wr1: (x_prescription.quantity_type :=: quantity_type) AND
       (y_prescription.quantity_type :=: quantity_type) AND
       (z_prescription.quantity_type :=: quantity_type);
END_ENTITY;
```

Attribute definitions:

— x_prescription prescribes the value for the translation in the x-direction
— y_prescription prescribes the value for the translation in the y-direction
— z_prescription prescribes the value for the translation in the z-direction
— x is derived from the literal or default value of x_prescription
— y is derived from the literal or default value of y_prescription
— z is derived from the literal or default value of z_prescription

Formal propositions:

— wr1: Each of the component prescriptions shall have the same quantity_type as the quantity_type of the **mgm_parametric_translation** itself.

4.3.5.11 ENTITY mgm_rotation

An **mgm_rotation** is a type of mgm_translation_or_rotation that specifies a rotation relative to a given rotation axis. It is an abstract supertype that provides a generic mechanism to reference an mgm_rotation_with_axes_fixed or an mgm_rotation_with_axes_moving.

NOTE An **mgm_rotation** is only used in an mgm_axis_transformation_sequence.

Express specification:


```
ENTITY mgm_rotation
  ABSTRACT SUPERTYPE OF(ONEOF(
    mgm_rotation_with_axes_fixed,
    mgm_rotation_with_axes_moving))
  SUBTYPE OF(mgm_translation_or_rotation);
  axis          : mgm_3d_direction;
  angle         : REAL;
  quantity_type : nrf_real_quantity_type;
WHERE
  wr1: quantity_type.quantity_category.name = 'plane_angle';
  wr2: {-360.0 <= angle <= 360.0};
END_ENTITY;
```

Attribute definitions:

- axis specifies an mgm_3d_direction, which is expressed in the reference system, indicating the orientation of the rotation axis. For operations the normalized version of axis has to be used.
- angle specifies the value of the angle over which the local system is rotated with respect to the reference system. angle is expressed in the unit of the applicable plane_angle_quantity_type.

Formal propositions:

- wr1: The quantity_type shall be a 'plane_angle'.
- wr1: angle shall be greater or equal to -360 and less or equal to +360.

4.3.5.12 ENTITY mgm_rotation_with_axes_fixed

An **mgm_rotation_with_axes_fixed** specifies a coordinate system rotation in which the reference system for any of the subsequent tranformations (either rotation of translation) in an mgm_axis_transformation_sequence remains unchanged.

NOTE An **mgm_rotation_with_axes_fixed** is only used in an mgm_axis_transformation_sequence.

Express specification:

```
ENTITY mgm_rotation_with_axes_fixed
  SUPERTYPE OF(ONEOF(mgm_parametric_rotation_with_axes_fixed))
  SUBTYPE OF(mgm_rotation);
END_ENTITY;
```

4.3.5.13 ENTITY mgm_parametric_rotation_with_axes_fixed

An **mgm_parametric_rotation_with_axes_fixed** is a type of mgm_rotation_with_axes_fixed that specifies its rotation angle with a value prescription. This enables the parametric definition of such a rotation.

Express specification:

```
ENTITY mgm_parametric_rotation_with_axes_fixed
  SUBTYPE OF(mgm_rotation_with_axes_fixed);
  angle_prescription : nrf_real_quantity_value_prescription;
DERIVE
  SELF\mgm_rotation.angle : REAL := angle_prescription.val;
WHERE
  wr1: angle_prescription.quantity_type :=: quantity_type;
END_ENTITY;
```


Attribute definitions:

- angle_prescription prescibes the value for the rotation angle.
- angle is derived from the literal or default value of angle_prescription.

Formal propositions:

- wr1: The angle_prescription shall have the same quantity_type as the quantity_type of the **mgm_parametric_rotation_with_axes_fixed** itself.

4.3.5.14 ENTITY mgm_rotation_with_axes_moving

An **mgm_rotation_with_axes_moving** defines a coordinate system rotation in which the reference system for any of the subsequent tranformations (either rotation of translation) in an mgm_axis_transformation_sequence is rotated according to **mgm_rotation_with_axes_moving**.

NOTE An **mgm_rotation_with_axes_moving** is only used in an mgm_axis_transformation_sequence.

Express specification:

```
ENTITY mgm_rotation_with_axes_moving
  SUPERTYPE OF(ONEOF(mgm_parametric_rotation_with_axes_moving))
  SUBTYPE OF(mgm_rotation);
END_ENTITY;
```

4.3.5.15 ENTITY mgm_parametric_rotation_with_axes_moving

An **mgm_parametric_rotation_with_axes_moving** is a type of mgm_rotation_with_axes_moving that specifies its rotation angle with a value prescription. This enables the parametric definition of such a rotation.

Express specification:

```
ENTITY mgm_parametric_rotation_with_axes_moving
  SUBTYPE OF(mgm_rotation_with_axes_moving);
  angle_prescription : nrf_real_quantity_value_prescription;
DERIVE
  SELF\mgm_rotation.angle : REAL := angle_prescription.val;
WHERE
  wr1: angle_prescription.quantity_type :=: quantity_type;
END_ENTITY;
```

Attribute definitions:

- angle_prescription prescibes the value for the rotation angle.
- angle is derived from the literal or default value of angle_prescription.

Formal propositions:

- wr1: The angle_prescription shall have the same quantity_type as the quantity_type of the **mgm_parametric_rotation_with_axes_moving** itself.

4.3.5.16 ENTITY mgm_quantity_context

An **mgm_quantity_context** specifies global quantity types and numerical uncertainty values for a model. The global quantity types (with units) apply to all parametric attributes of the instances that define the model. The actual global quantity types and uncertainties are defined in an external dictionary. An **mgm_quantity_context** shall as a minimum include the following quantity types: 'length', 'plane_angle', 'time', 'temperature', 'velocity' and 'angular_velocity', and the following uncertainty: 'point_coincidence_length'.

NOTE 1 The tolerance to be used to determine whether or not two angles are considered to be coincident is derived from the 'point_coincidence_length' uncertainty. This angle tolerance, expressed in radians, is the angle needed to rotate an object at a distance of one length unit such that the object is translated along a distance equal to an arc-length equal to 'point_coincidence_length' uncertainty. This implies that the value of the angle tolerance expressed in radians is equal to the value of 'point_coincidence_length' uncertainty expressed in the length unit.

NOTE 2 An **mgm_quantity_context** is referenced from an mgm_meshed_geometric_model. The **mgm_quantity_context** is only applicable to the instances contained within the mgm_meshed_geometric_model. Any submodels contained in the model shall reference the same **mgm_quantity_context** instance, thereby ensuring that all defining instances in a model conform to one global quantity and unit context. A different mgm_meshed_geometric_model model/submodel tree (i.e. with a different root model) may reference a different **mgm_quantity_context** instance.

Express specification:

```
ENTITY mgm_quantity_context;
  quantity_types : LIST OF nrf_any_quantity_type;
  uncertainties  : LIST OF nrf_real_quantity_value_literal;
WHERE
  wr1: mgm_verify_context_quantity_types(SELF);
  wr2: mgm_verify_context_uncertainties(SELF);
END_ENTITY;
```

Attribute definitions:

- quantity_types specifies the global quantity types (with units) that apply to all parametric attributes of the instances in a model that makes an **mgm_quantity_context** applicable.
- uncertainties specifies the uncertainty values or numerical tolerances for a model that makes an **mgm_quantity_context** applicable.

Formal propositions:

- wr1: The quantity_types shall have unique quantity_category names and shall include as a minimum: 'length', 'plane_angle', 'time', 'temperature', 'velocity' and 'angular_velocity'.
- wr2: The uncertainties shall as a minimum include a 'point_coincidence_length' specified in the same unit as the context's 'length' and greater than zero. The 'point_coincidence_length' is the distance within which two points are considered to be at the same location.

4.3.5.17 FUNCTION mgm_verify_context_quantity_types

The function **mgm_verify_context_quantity_types** verifies that the names of the quantity categories referenced in the quantity_types of an mgm_quantity_context are unique and that as a minimum the

following quantity types are included: 'length', 'plane_angle', 'time', 'temperature', 'velocity' and 'angular_velocity'.

Express specification:

```
FUNCTION mgm_verify_context_quantity_types(
    a_quantity_context : mgm_quantity_context) : BOOLEAN;
  LOCAL
    qc : nrf_physical_quantity_category;
    qc_names : LIST OF nrf_non_blank_label := [];
    length_verified : BOOLEAN := FALSE;
    plane_angle_verified : BOOLEAN := FALSE;
    time_verified : BOOLEAN := FALSE;
    temperature_verified : BOOLEAN := FALSE;
    velocity_verified : BOOLEAN := FALSE;
    angular_velocity_verified : BOOLEAN := FALSE;
  END_LOCAL;
  REPEAT i := 1 TO SIZEOF(a_quantity_context.quantity_types);
    qc := a_quantity_context.quantity_types[i].quantity_category;
    IF qc.name IN qc_names THEN
      -- non-unique name
      RETURN(FALSE);
    END_IF;
    qc_names := qc_names + qc.name;
    CASE qc.name OF
      'length': length_verified := nrf_verify_dimensional_exponents(
        qc, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
      'plane_angle': plane_angle_verified := nrf_verify_dimensional_exponents(
        qc, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
      'time': time_verified := nrf_verify_dimensional_exponents(
        qc, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0);
      'temperature': temperature_verified := nrf_verify_dimensional_exponents(
        qc, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0);
      'velocity': velocity_verified := nrf_verify_dimensional_exponents(
        qc, 1.0, 0.0, -1.0, 0.0, 0.0, 0.0, 0.0);
      'angular_velocity': angular_velocity_verified := nrf_verify_dimensional_exponents(
        qc, 0.0, 0.0, -1.0, 0.0, 0.0, 0.0, 0.0);
    END_CASE;
  END_REPEAT;
  IF NOT (length_verified AND
      plane_angle_verified AND
      time_verified AND
      temperature_verified AND
      velocity_verified AND
      angular_velocity_verified) THEN
    RETURN(FALSE);
  END_IF;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— a_quantity_context specifies the mgm_quantity_context to be verified.

4.3.5.18 FUNCTION mgm_verify_context_uncertainties

The function **mgm_verify_context_uncertainties** verifies that an mgm_quantity_context has at least an uncertainty specified for a 'point_coincidence_length' and that it uses the same unit as the context's 'length' quantity type.

Express specification:

```
FUNCTION mgm_verify_context_uncertainties(
    a_quantity_context : mgm_quantity_context) : BOOLEAN;
LOCAL
  valid : BOOLEAN := FALSE;
  length_quantity_type : nrf_any_quantity_type;
END_LOCAL;
  REPEAT i := 1 TO SIZEOF(a_quantity_context.quantity_types);
    IF a_quantity_context.quantity_types[i].quantity_category.name = 'length' THEN
      length_quantity_type := a_quantity_context.quantity_types[i];
      ESCAPE;
    END_IF;
  END_REPEAT;
  REPEAT i := 1 TO SIZEOF(a_quantity_context.uncertainties);
    IF a_quantity_context.uncertainties[i].quantity_type.name =
        'point_coincidence_length' THEN
      IF a_quantity_context.uncertainties[i].quantity_type.unit.name =
          length_quantity_type.unit.name THEN
        valid := TRUE;
      END_IF;
    END_IF;
  END_REPEAT;
  RETURN(valid);
END_FUNCTION;
```

Argument definitions:

— a_quantity_context specifies the mgm_quantity_context to be verified.

4.3.5.19 FUNCTION mgm_get_context_quantity_type

The function **mgm_get_context_quantity_type** returns the nrf_any_quantity_type from the quantity_types in the quantity_context of a given mgm_meshed_geometric_model for a given basic quantity type name. If no match can be found the function returns indeterminate.

Express specification:

```
FUNCTION mgm_get_context_quantity_type(
    a_model : mgm_meshed_geometric_model;
    a_quantity_category_name : STRING) : nrf_any_quantity_type;
  REPEAT i := 1 TO SIZEOF(a_model.quantity_context.quantity_types);
    IF a_model.quantity_context.quantity_types[i].quantity_category.name =
        a_quantity_category_name THEN
      RETURN(a_model.quantity_context.quantity_types[i]);
    END_IF;
  END_REPEAT;
  RETURN(?);
END_FUNCTION;
```

Argument definitions:

— a_model specifies the mgm_meshed_geometric_model from which to get the quantity_context.

— a_quantity_category_name specifies the name to match the name of the quantity_category of the nrf_any_quantity_type to be retrieved.

4.3.5.20 FUNCTION mgm_get_context_uncertainty_value

The function **mgm_get_context_uncertainty_value** returns the value of the nrf_real_quantity_value_literal from the uncertainties in the quantity_context of a given mgm_meshed_geometric_model, for which the name of the quantity_type matches a given name. If no match can be found the function returns indeterminate.

Express specification:

```
FUNCTION mgm_get_context_uncertainty_value(
    a_model : mgm_meshed_geometric_model;
    an_uncertainty_name : STRING) : REAL;
  REPEAT i := 1 TO SIZEOF(a_model.quantity_context.uncertainties);
    IF a_model.quantity_context.uncertainties[i].quantity_type.name =
        an_uncertainty_name THEN
 RETURN(a_model.quantity_context.uncertainties[i].val);
    END_IF;
  END_REPEAT;
  RETURN(?);
END_FUNCTION;
```

Argument definitions:

- a_model specifies the mgm_meshed_geometric_model from which to get the uncertainty value.
- an_uncertainty_name specifies the name to match the name of the quantity_type of the nrf_real_quantity_value_literal of which to return the value.

4.3.5.21 FUNCTION mgm_compute_distance_between_points

The function **mgm_compute_distance_between_points** computes and returns the distance between two points of which the locations are defined by mgm_3d_cartesian_point instances. The distance is expressed in the length unit of the coordinates of the two points.

Express specification:

```
FUNCTION mgm_compute_distance_between_points(
    p1 : mgm_3d_cartesian_point;
    p2 : mgm_3d_cartesian_point): REAL;

  LOCAL
    dx, dy, dz : REAL;
  END_LOCAL;
  dx := p2.x - p1.x;
  dy := p2.y - p1.y;
  dz := p2.z - p1.z;
  RETURN(SQRT(dx*dx + dy*dy + dz*dz));

END_FUNCTION;
```

Argument definitions:

- p1 specifies the first mgm_3d_cartesian_point
- p2 specifies the second mgm_3d_cartesian_point

4.3.5.22 RULE mgm_all_plane_angle_quantity_types_in_degree

The RULE **mgm_all_plane_angle_quantity_types_in_degree** requires that all nrf_real_quantity instances in the dataset that represent a 'plane_angle' use 'degree' as a unit.

Express specification:

```
RULE mgm_all_plane_angle_quantity_types_in_degree FOR (nrf_real_quantity_type);
LOCAL
  rule_satisfied : LOGICAL := TRUE;
END_LOCAL;
  REPEAT i := 1 TO SIZEOF(nrf_real_quantity_type) WHILE rule_satisfied;
    IF nrf_real_quantity_type[i].quantity_category.name = 'plane_angle' THEN
      IF nrf_real_quantity_type[i].unit.name <> 'degree' THEN
        rule_satisfied := FALSE;
      END_IF;
    END_IF;
  END_REPEAT;
WHERE
  wr1: rule_satisfied;
END_RULE;
```

4.3.6 MGM meshed geometric model UoF

The MGM meshed geometric model UoF contains the application objects to define a geometric model consisting of surfaces meshed with oriented faces for the purpose of engineering analysis using shell geometry.

4.3.6.1 TYPE mgm_active_side_type

An **mgm_active_side_type** is an enumeration that indicates the activity type of an mgm_meshed_primitive_bounded_surface. An mgm_meshed_primitive_bounded_surface has two sides, side 1 and side 2 that are defined by the mgm_primitive_bounded_surface that is associated through the surface attribute of the mgm_meshed_primitive_bounded_surface.

Express specification:

```
TYPE mgm_active_side_type = ENUMERATION OF(
  NONE,
  SIDE1,
  SIDE2,
  BOTH);
END_TYPE;
```

Enumeration value definitions:

SIDE1: indicates that the faces on side 1 of an mgm_meshed_primitive_bounded_surface are active and the faces on side 2 are inactive.

SIDE2: indicates that the faces on side 2 of an mgm_meshed_primitive_bounded_surface are active and the faces on side 1 are inactive.

BOTH: indicates that the faces on both sides of an mgm_meshed_primitive_bounded_surface are active.

NONE: indicates that the faces on both sides of an mgm_meshed_primitive_bounded_surface are inactive.

NOTE 1 For thermal-radiative applications inactive faces do not emit nor reflect thermal radiation, but act as shaders in the thermal-radiative model, with an artificial absorptance of 1.0 for all thermal radiation.

NOTE 2 mgm_meshed_primitive_bounded_surface instances with active_side = NONE shall not have associated side1_surface_material and side2_surface_material.

4.3.6.2 ENTITY mgm_meshed_geometric_model

An **mgm_meshed_geometric_model** is a type of nrf_network_model specifying a geometric model representation consisting of nodes that are mgm_any_meshed_geometric_item instances. These may be collected in a hierarchical tree structure using mgm_compound_meshed_geometric_item instances. Any static or dynamic (kinematic) coordinate system transformations may be specified at each level for each individual mgm_any_meshed_geometric_item in the tree structure.

The geometric items are sets of oriented, bounded faces for the purpose of engineering analysis using shell geometry. The faces are associated with mgm_primitive_bounded_surface instances in mgm_meshed_primitive_bounded_surface instances, which may contain various types of meshing.

Express specification:

```
ENTITY mgm_meshed_geometric_model
  SUBTYPE OF(nrf_network_model);
  SELF\nrf_network_model.nodes     : LIST OF UNIQUE mgm_any_meshed_geometric_item;
  SELF\nrf_network_model.submodels : LIST OF UNIQUE mgm_meshed_geometric_model;
  quantity_context                 : mgm_quantity_context;
  root_item                        : mgm_any_meshed_geometric_item;
  enclosures                       : LIST OF UNIQUE mgm_enclosure;
WHERE
  wr1: SIZEOF(node_relationships) = 0;
  wr2: SIZEOF(QUERY(sm <* submodels | sm.quantity_context :=: SELF.quantity_context)) =
    SIZEOF(submodels);
  wr3: mgm_verify_acyclic_compound_meshed_geometric_item_tree(root_item, [root_item]);
END_ENTITY;
```

Attribute definitions:

- SELF\nrf_network_model.nodes specifies all instances, which are a subtype of mgm_any_meshed_geometric_item, that are contained in an **mgm_meshed_geometric_model**.
- SELF\nrf_network_model.submodels specifies the next lower level **mgm_meshed_geometric_model** instances that are part of an **mgm_meshed_geometric_model**
- quantity_context specifies the mgm_quantity_context defining the applicable quantity types, units and uncertainties for the model
- root_item specifies the mgm_any_meshed_geometric_item that is the root of the mgm_compound_meshed_geometric_item tree
- enclosures specifies the list of enclosures defined for the model, if any.

Formal propositions:

- wr1: No nrf_network_node_relationships shall be defined for an **mgm_meshed_geometric_model**.
- wr2: The quantity_context of all submodels shall be the same as the quantity_context of the containing **mgm_meshed_geometric_model**. This implies that all **mgm_meshed_geometric_model** instances in one submodel tree shall use the same mgm_quantity_context.

— wr3: If the root_item comprises an mgm_compound_meshed_geometric_item tree, it shall be an acyclic tree.

4.3.6.3 ENTITY mgm_any_meshed_geometric_item

An **mgm_any_meshed_geometric_item** is a type of nrf_network_node that provides a generic mechanism to reference to instances of mgm_compound_meshed_geometric_item, mgm_meshed_geometric_item_by_submodel, mgm_meshed_primitive_bounded_surface or mgm_meshed_boolean_difference_surface. It is the construction object for an mgm_meshed_geometric_model. Each **mgm_any_meshed_geometric_item** has an optional transformation that can be used to locate and orient it with respect to its next higher level containing geometric item or model. Any **mgm_any_meshed_geometric_item** may be articulated during a run of a model through the specification of kinematic_articulations in an sma_space_mission_case. Such kinematic movement is applied in addition to the static transformation.

Express specification:

```
ENTITY mgm_any_meshed_geometric_item
  ABSTRACT SUPERTYPE OF(ONEOF(
    mgm_compound_meshed_geometric_item,
    mgm_meshed_geometric_item_by_submodel,
    mgm_meshed_primitive_bounded_surface,
    mgm_meshed_boolean_difference_surface))
  SUBTYPE OF(nrf_network_node);
  transformation : OPTIONAL mgm_axis_transformation;
WHERE
  wr1: mgm_verify_transformation(SELF);
END_ENTITY;
```

Attribute definitions:

— transformation optionally specifies an mgm_axis_transformation that defines a static coordinate system transformation to be applied relative to the reference system of the next higher level **mgm_any_meshed_geometric_item** or mgm_meshed_geometric_model.

Formal propositions:

— wr1: The transformation shall be valid.

4.3.6.4 ENTITY mgm_compound_meshed_geometric_item

An **mgm_compound_meshed_geometric_item** is a type of mgm_any_meshed_geometric_item that specifies a collection of mgm_any_meshed_geometric_item instances in a list. It provides the mechanism to define the geometric items of an mgm_meshed_geometric_model in a hierarchical tree, where at each level coordinate transformations may be applied.

Express specification:

```
ENTITY mgm_compound_meshed_geometric_item
  SUPERTYPE OF(ONEOF(mgm_qualified_compound_meshed_primitive_bounded_surface))
  SUBTYPE OF(mgm_any_meshed_geometric_item);
  geometric_items : LIST [1:?] OF UNIQUE mgm_any_meshed_geometric_item;
END_ENTITY;
```


Attribute definitions:

— geometric_items specifies the list of mgm_any_meshed_geometric_item instances to be collected into an **mgm_compound_meshed_geometric_item**.

4.3.6.5 ENTITY mgm_meshed_geometric_item_by_submodel

An **mgm_meshed_geometric_item_by_submodel** is a type of mgm_any_meshed_geometric_item that specifies an mgm_meshed_geometric_model (as a submodel) for inclusion at some level into a hierarchical tree of mgm_compound_meshed_geometric_item instances, possibly with a coordinate transformation.

Express specification:

```
ENTITY mgm_meshed_geometric_item_by_submodel
  SUBTYPE OF(mgm_any_meshed_geometric_item);
  submodel : mgm_meshed_geometric_model;
WHERE
  wr1: submodel IN containing_model.submodels;
END_ENTITY;
```

Attribute definitions:

— submodel specifies an mgm_meshed_geometric_model as a submodel that defines a complete set of bounded faces.

Formal propositions:

— wr1: The submodel shall be registered as a submodel of the containing model.

4.3.6.6 ENTITY mgm_meshed_primitive_bounded_surface

An **mgm_meshed_primitive_bounded_surface** is a type of mgm_any_meshed_geometric_item that specifies a connected set of faces defined on one or on both sides of an mgm_primitive_bounded_surface. It is by definition a leave node in a hierarchical mgm_compound_meshed_geometric_item tree.

Express specification:

```
ENTITY mgm_meshed_primitive_bounded_surface
  SUBTYPE OF(mgm_any_meshed_geometric_item);
  surface                  : mgm_primitive_bounded_surface;
  active_side              : mgm_active_side_type;
  side1_notional_thickness : OPTIONAL nrf_real_quantity_value_prescription;
  side2_notional_thickness : OPTIONAL nrf_real_quantity_value_prescription;
  side1_surface_material   : OPTIONAL nrf_material;
  side2_surface_material   : OPTIONAL nrf_material;
  side1_bulk_material      : OPTIONAL nrf_material;
  side2_bulk_material      : OPTIONAL nrf_material;
  side1_colour             : OPTIONAL mgm_colour_rgb;
  side2_colour             : OPTIONAL mgm_colour_rgb;
  dir1_meshing             : nrf_positive_integer;
  dir2_meshing             : nrf_positive_integer;
  dir1_submeshing          : OPTIONAL nrf_positive_integer;
  dir2_submeshing          : OPTIONAL nrf_positive_integer;
  dir1_grid_spacings       : OPTIONAL LIST [1:?] OF REAL;
  dir2_grid_spacings       : OPTIONAL LIST [1:?] OF REAL;
  side1_faces              : LIST [1:?] OF mgm_face;
```



```
  side2_faces              : LIST [1:?] OF mgm_face;
WHERE
  has_valid_side1_notional_thickness: (NOT EXISTS(side1_notional_thickness)) OR
    ((EXISTS(side1_notional_thickness) AND (side1_notional_thickness.quantity_type :=:
    mgm_get_context_quantity_type(containing_model, 'length')) AND
    (side1_notional_thickness.val > 0.0)));
  has_valid_side2_notional_thickness: (NOT EXISTS(side2_notional_thickness)) OR
    ((EXISTS(side2_notional_thickness) AND (side2_notional_thickness.quantity_type :=:
    mgm_get_context_quantity_type(containing_model, 'length')) AND
    (side2_notional_thickness.val > 0.0)));
  has_valid_side1_pseudo_solid:
    (EXISTS(side1_notional_thickness) AND EXISTS(side1_bulk_material)) OR
    (NOT EXISTS(side1_notional_thickness) AND NOT EXISTS(side1_bulk_material));
  has_valid_side2_pseudo_solid:
    (EXISTS(side2_notional_thickness) AND EXISTS(side2_bulk_material)) OR
    (NOT EXISTS(side2_notional_thickness) AND NOT EXISTS(side2_bulk_material));
  has_valid_dir1_grid_spacings: (NOT EXISTS(dir1_grid_spacings)) OR
    (EXISTS(dir1_grid_spacings) AND (SIZEOF(dir1_grid_spacings) = dir1_meshing+1) AND
    mgm_verify_surface_grid_spacings(dir1_grid_spacings));
  has_valid_dir2_grid_spacings: (NOT EXISTS(dir2_grid_spacings)) OR
    (EXISTS(dir2_grid_spacings) AND (SIZEOF(dir2_grid_spacings) = dir2_meshing+1) AND
    mgm_verify_surface_grid_spacings(dir2_grid_spacings));
  has_correct_number_of_side1_faces: SIZEOF(side1_faces) = dir1_meshing*dir2_meshing;
  has_correct_number_of_side2_faces: SIZEOF(side2_faces) = dir1_meshing*dir2_meshing;
  has_valid_active_side1: (((active_side = mgm_active_side_type.BOTH) OR
    (active_side = mgm_active_side_type.SIDE1)) AND EXISTS(side1_surface_material)) OR
    (active_side = mgm_active_side_type.NONE) OR
    (active_side = mgm_active_side_type.SIDE2);
  has_valid_active_side2: (((active_side = mgm_active_side_type.BOTH) OR
    (active_side = mgm_active_side_type.SIDE2)) AND EXISTS(side2_surface_material)) OR
    (active_side = mgm_active_side_type.NONE) OR
    (active_side = mgm_active_side_type.SIDE1);
  surface_has_correct_inverse_reference: surface.geometric_item :=: SELF;
END_ENTITY;
```

Attribute definitions:

- surface specifies the mgm_primitive_bounded_surface defining the geometry of an **mgm_meshed_primitive_bounded_surface**
- active_side specifies one item of mgm_active_side_type that is an enumeration type.
- side1_notional_thickness optionally specifies a notional thickness that may be used to derive a 'pseudo-solid' object for analysis purposes. This 'pseudo-solid' is assumed to be located at side 1 of surface and the thickness is measured from the plane of surface.
- side2_notional_thickness optionally specifies a notional thickness that may be used to derive a 'pseudo-solid' object for analysis purposes. This 'pseudo-solid' is assumed to be located at side 2 of surface and the thickness is measured from the plane of surface.
- side1_surface_material optionally specifies the nrf_material that defines the surface property values of side 1 of surface.
- side2_surface_material optionally specifies the nrf_material that defines the surface property values of side 2 of surface.
- side1_bulk_material optionally specifies the nrf_material that defines the bulk property values of the pseudo-solid created on side 1 of the surface through specification of side1_notional_thickness.
- side2_bulk_material optionally specifies the nrf_material that defines the bulk property values of the pseudo-solid created on side 2 of the surface through specification of side1_notional_thickness.
- side1_colour optionally specifies the mgm_colour_rgb to be used for visualization of side 1 of an **mgm_meshed_primitive_bounded_surface**.

- side2_colour optionally specifies the mgm_colour_rgb to be used for visualization of side 2 of an **mgm_meshed_primitive_bounded_surface**.
- dir1_meshing specifies the number of mesh-divisions, being at least 1, along direction-1 of surface.
- dir2_meshing specifies the number of mesh-divisions, being at least 1, along direction-2 of surface.
- dir1_submeshing optionally specifies the number of sub-mesh- divisions, being at least 1, along direction 1 of surface to be applied to each face meshed on surface.
- dir2_submeshing optionally specifies the number of sub-mesh- divisions, being at least 1, along direction 2 of surface to be applied to each face meshed on surface.
- dir1_grid_spacings optionally specifies a list of coordinates in direction-1 of surface location of the face boundaries. If dir1_grid_spacings is not specified a uniform grid spacing is implied along direction-1 of surface.
- dir2_grid_spacings optionally specifies a list of coordinates in direction-2 of surface location of the face boundaries. If dir2_grid_spacings is not specified a uniform grid spacing is implied along direction-2 of surface.
- side1_faces specifies the list of faces meshed on side 1 of the surface. The ordering of the faces in the list is first along direction-1 and then along direction-2 as defined for the associated surface.
- side2_faces specifies the list of faces meshed on side 2 of the surface. The ordering of the faces in the list is first along direction-1 and then along direction-2 as defined for the associated surface.

Formal propositions:

- has_valid_side1_notional_thickness: If side1_notional_thickness is specified, it shall have the applicable 'length' quantity_type and a value greater than zero.
- has_valid_side2_notional_thickness: If side2_notional_thickness is specified, it shall have the applicable 'length' quantity_type and a value greater than zero.
- has_valid_side1_pseudo_solid: If side1_notional_thickness is specified, then also side1_bulk_material must be specified.
- has_valid_side2_pseudo_solid: If side2_notional_thickness is specified, then also side2_bulk_material must be specified.
- has_valid_dir1_grid_spacings: If dir1_grid_spacings is specified, the number of coordinate values must be equal to dir1_meshing+1 and the values of dir1_grid_spacings must be in ascending order and start with 0.0 and ends with 1.0.
- has_valid_dir2_grid_spacings: If dir2_grid_spacings is specified, the number of coordinate values must be equal to dir2_meshing+1 and the values of dir2_grid_spacings must be in ascending order and start with 0.0 and ends with 1.0.
- has_correct_number_of_side1_faces: The number of side1_faces shall be equal to the product of dir1_meshing and dir2_meshing.
- has_correct_number_of_side2_faces: The number of side2_faces shall be equal to the product of dir1_meshing and dir2_meshing.
- has_valid_active_side1: If active_side is SIDE1 or BOTH then side1_surface_material must be specified.
- has_valid_active_side2: If active_side is SIDE2 or BOTH then side2_surface_material must be specified.
- surface_has_correct_inverse_reference: The geometric_item of surface shall reference this **mgm_meshed_primitive_bounded_surface**.

4.3.6.7 RULE mgm_verify_referencing_of_meshed_geometric_items

The RULE **mgm_verify_referencing_of_meshed_geometric_items** verifies that any mgm_any_meshed_geometric_item that is part of any mgm_meshed_geometric_model is referenced only once.

Express specification:

```
RULE mgm_verify_referencing_of_meshed_geometric_items FOR (nrf_root);
LOCAL
  rule_satisfied : LOGICAL := TRUE;
  all_items : LIST OF mgm_any_meshed_geometric_item := [];
  root_model : nrf_network_model;
  check_item : mgm_any_meshed_geometric_item;
END_LOCAL;
  REPEAT i := 1 TO SIZEOF(nrf_root[1].root_models) WHILE rule_satisfied;
    root_model := nrf_root[1].root_models[i];
    IF 'MGM_ARM.MGM_MESHED_GEOMETRIC_MODEL' IN TYPEOF(root_model) THEN
      all_items := all_items +
 mgm_get_items_from_meshed_geometric_item(root_model.root_item);
    END_IF;
  END_REPEAT;
  REPEAT WHILE ((SIZEOF(all_items) > 1) AND rule_satisfied);
    check_item := all_items[1];
    REMOVE(all_items, 1);
    IF check_item IN all_items THEN
      rule_satisfied := FALSE;
    END_IF;
  END_REPEAT;
WHERE
  wr1: rule_satisfied;
END_RULE;
```

4.3.6.8 FUNCTION mgm_get_items_from_meshed_geometric_item

The function **mgm_get_items_from_meshed_geometric_item** returns a list of all mgm_any_meshed_geometric_item instances that are part of a given mgm_any_meshed_geometric_item.

Express specification:

```
FUNCTION mgm_get_items_from_meshed_geometric_item (
    an_item : mgm_any_meshed_geometric_item) : LIST OF mgm_any_meshed_geometric_item;
  LOCAL
    all_items : LIST OF mgm_any_meshed_geometric_item := [];
  END_LOCAL;
  all_items := all_items + [an_item];
  IF 'MGM_ARM.MGM_COMPOUND_MESHED_GEOMETRIC_ITEM' IN TYPEOF(an_item) THEN
    REPEAT i := 1 TO SIZEOF(an_item.geometric_items);
      all_items := all_items +
 mgm_get_items_from_meshed_geometric_item(an_item.geometric_items[i]);
    END_REPEAT;
  ELSE
    IF 'MGM_ARM.MGM_MESHED_BOOLEAN_DIFFERENCE_SURFACE' IN TYPEOF(an_item) THEN
      all_items := all_items +
 mgm_get_items_from_meshed_geometric_item(an_item.base_surface);
    ELSE
      IF 'MGM_ARM.MGM_MESHED_GEOMETRIC_ITEM_BY_SUBMODEL' IN TYPEOF(an_item) THEN
        all_items := all_items +
 mgm_get_items_from_meshed_geometric_item(an_item.submodel.root_item);
      END_IF;
```

```
    END_IF;
  END_IF;
  RETURN(all_items);
END_FUNCTION;
```

Argument definitions:

— an_item specifies the mgm_any_meshed_geometric_item for which all contained mgm_any_meshed_geometric_item instances should be returned.

4.3.6.9 ENTITY mgm_primitive_bounded_surface

An **mgm_primitive_bounded_surface** is a surface of finite area with identifiable boundaries. It is a generic object that specifies the attributes common to all primitive bounded surfaces.

The side of an **mgm_primitive_bounded_surface** that corresponds to the side with the positive normal is referred to as side 1 and the opposite side is referred to as side 2.

Express specification:

```
ENTITY mgm_primitive_bounded_surface
  ABSTRACT SUPERTYPE OF(ONEOF(
    mgm_triangle,
    mgm_rectangle,
    mgm_quadrilateral,
    mgm_disc,
    mgm_cylinder,
    mgm_cone,
    mgm_sphere,
    mgm_paraboloid));
  geometric_item : mgm_any_meshed_geometric_item;
END_ENTITY;
```

Attribute definitions:

— geometric_item specifies the mgm_any_meshed_geometric_item that is using this **mgm_primitive_bounded_surface** to define its surface shape.

4.3.6.10 ENTITY mgm_triangle

An **mgm_triangle** is a type of mgm_primitive_bounded_surface that specifies a planar surface bounded by three straight edges defined by three points.

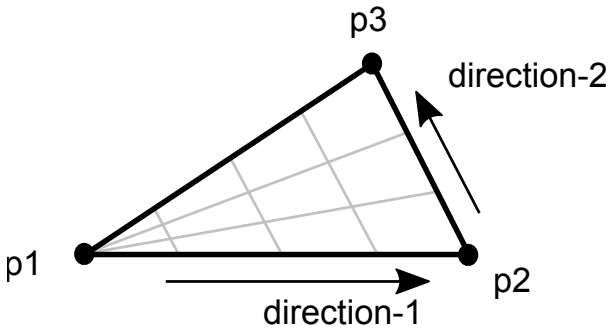

Figure 1 Sketch of **mgm_triangle**

The positive normal on the surface is defined by the cross product of the vector from p1 to p2 and the vector from p1 to p3.

The first parametric direction (direction-1) is defined along the vector from p1 to p2, the second parametric direction (direction-2) is defined along the vector from p2 to p3.

Express specification:

```
ENTITY mgm_triangle
  SUBTYPE OF(mgm_primitive_bounded_surface);
  p1 : mgm_3d_cartesian_point;
  p2 : mgm_3d_cartesian_point;
  p3 : mgm_3d_cartesian_point;
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2, p3], geometric_item.containing_model);
  wr2: mgm_verify_no_coincident_points([p1, p2, p3], mgm_get_context_uncertainty_value(
    geometric_item.containing_model, 'point_coincidence_length'));
  wr3: mgm_verify_no_colinear_points([p1, p2, p3], mgm_get_context_uncertainty_value(
    geometric_item.containing_model, 'point_coincidence_length'));
END_ENTITY;
```

Attribute definitions:

- p1 specifies an mgm_3d_cartesian_point that defines the first vertex of the triangle.
- p2 specifies an mgm_3d_cartesian_point that defines the second vertex of the triangle.
- p3 specifies an mgm_3d_cartesian_point that defines the third vertex of the triangle.

Formal propositions:

- wr1: the coordinates of all points shall all use the length_quantity_type specified in the global_quantity_context of the containing model.
- wr2: all edges of the triangle shall have a length greater than the point_coincidence_length uncertainty.
- wr3: the three points shall not be co-linear.

4.3.6.11 ENTITY mgm_rectangle

An **mgm_rectangle** is a type of mgm_primitive_bounded_surface that specifies a planar rectangular bounded surface defined by three points.

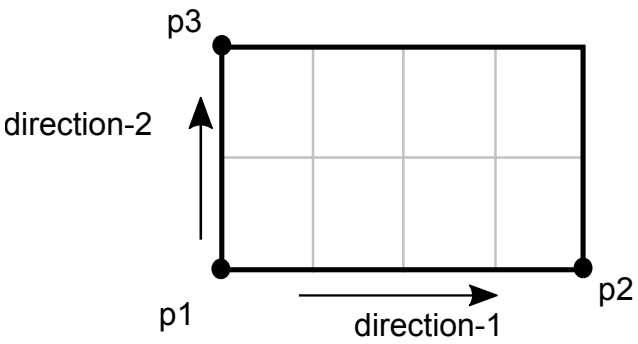

Figure 2 Sketch of an **mgm_rectangle**

The positive normal on the surface is defined by the cross product of the vector from p1 to p2 and the vector from p1 to p3.

The first parametric direction (direction-1) is defined along the vector from p1 to p2, the second parametric direction (direction-2) is defined along the vector from p1 to p3.

Points p1, p2 and p3 define three vertices of the rectangular bounded surface. The vectors from p1 to p2 and from p1 to p3 are required to be orthogonal. The fourth vertex of the rectangular surface is implied to be located by the vector sum of the vector to p2 plus the vector from p1 to p3.

Express specification:

```
ENTITY mgm_rectangle
  SUBTYPE OF(mgm_primitive_bounded_surface);
  p1 : mgm_3d_cartesian_point;
  p2 : mgm_3d_cartesian_point;
  p3 : mgm_3d_cartesian_point;
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2, p3], geometric_item.containing_model);
  wr2: mgm_verify_points_span_orthogonal_system(p1, p2, p3,
    mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
END_ENTITY;
```

Attribute definitions:

- p1 specifies an mgm_3d_cartesian_point that defines the first vertex of the rectangle.
- p2 specifies an mgm_3d_cartesian_point that defines the second vertex of the rectangle.
- p3 specifies an mgm_3d_cartesian_point that defines the third vertex of the rectangle.

Formal propositions:

- wr1: the coordinates of points p1, p2, p3 shall all use the length_quantity_type specified in the quantity_context of the containing model.
- wr2: points p1, p2 and p3 shall span an orthogonal system.

4.3.6.12 ENTITY mgm_quadrilateral

An **mgm_quadrilateral** is a type of mgm_primitive_bounded_surface that specifies a planar quadrilateral surface bounded by four straight edges, defined by four points.

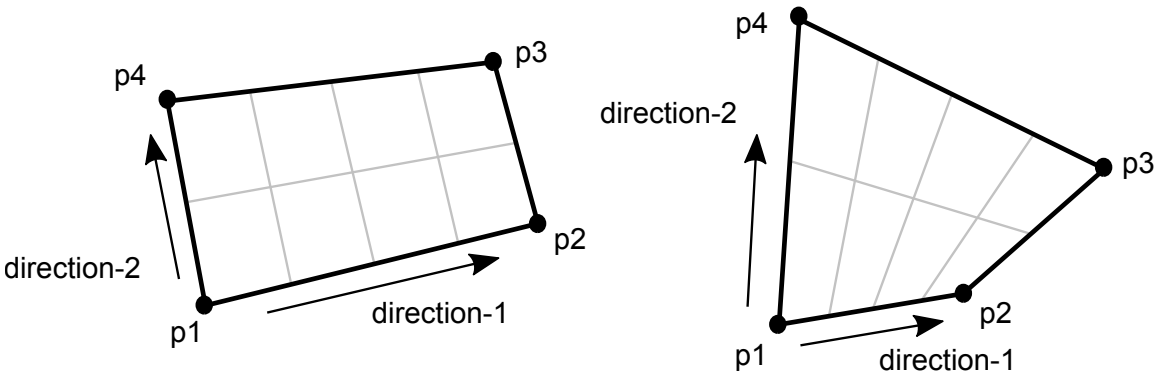

Figure 3 Sketch of two **mgm_quadrilateral** bounded surfaces

The positive normal on the surface is defined by the cross product of the vector from p1 to p2 and the vector from p1 to p4.

The first parametric direction (direction-1) is defined along the vector from p1 to p2, the second parametric direction (direction-2) is defined along the vector from p1 to p4.

Express specification:

```
ENTITY mgm_quadrilateral
  SUBTYPE OF(mgm_primitive_bounded_surface);
  p1 : mgm_3d_cartesian_point;
  p2 : mgm_3d_cartesian_point;
  p3 : mgm_3d_cartesian_point;
  p4 : mgm_3d_cartesian_point;
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2, p3, p4], geometric_item.containing_model);
  wr2: mgm_verify_quadrilateral(SELF);
END_ENTITY;
```

Attribute definitions:

- p1 specifies an mgm_3d_cartesian_point that defines the first vertex of the quadrilateral.
- p2 specifies an mgm_3d_cartesian_point that defines the second vertex of the quadrilateral.
- p3 specifies an mgm_3d_cartesian_point that defines the third vertex of the quadrilateral.
- p4 specifies an mgm_3d_cartesian_point that defines the fourth vertex of the quadrilateral.

Formal propositions:

- wr1: the coordinates of points p1, p2, p3, p4 shall all use the length_quantity_type specified in the quantity_context of the containing model.
- wr2: Verify the following constraints: (1) The distance between each pair of points shall be greater than the applicable 'point_coincidence_length', (2) The quadrilateral shall be planar, (3) All four interior angles of the quadrilateral shall be greater than 0 and less than 180 degrees.

4.3.6.13 ENTITY mgm_disc

An **mgm_disc** is a type of mgm_primitive_bounded_surface that defines a disc, ring, sector of disc or sector of ring.

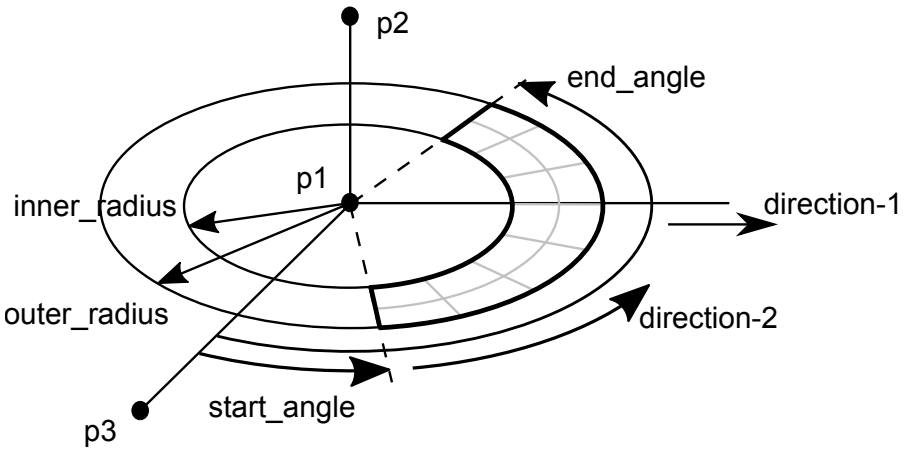

Figure 4 Sketch of **mgm_disc**

The vector from p1 to p2 defines the positive normal on the surface.

The first parametric direction (direction-1) is defined along the radius of the **mgm_disc**, the second parametric direction (direction-2) is defined along the positive direction of revolution about the vector from p1 to p2.

Express specification:

```
ENTITY mgm_disc
  SUBTYPE OF(mgm_primitive_bounded_surface);
  p1           : mgm_3d_cartesian_point;
  p2           : mgm_3d_cartesian_point;
  p3           : mgm_3d_cartesian_point;
  inner_radius : nrf_real_quantity_value_prescription;
  outer_radius : nrf_real_quantity_value_prescription;
  start_angle  : nrf_real_quantity_value_prescription;
  end_angle    : nrf_real_quantity_value_prescription;
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2, p3], geometric_item.containing_model);
  wr2: (inner_radius.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'))
    AND (outer_radius.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'));
  wr3: (start_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'))
    AND (end_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'));
  wr4: mgm_verify_points_span_orthogonal_system(p1, p2, p3,
    mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
  wr5: inner_radius.val >= 0.0;
  wr6: outer_radius.val > inner_radius.val +
    mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length');
  wr7: mgm_verify_start_and_end_angles(
    start_angle.val, end_angle.val, mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
END_ENTITY;
```

Attribute definitions:

- p1 specifies an mgm_3d_cartesian_point that is the centre of the disc and defines together with p2 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2.
- p2 specifies an mgm_3d_cartesian_point that defines together with p1 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2.
- p3 specifies an mgm_3d_cartesian_point that defines together with p1 and p2 the reference plane from which start_angle and end_angle are measured.
- inner_radius specifies the value of the inner radius in the case an **mgm_disc** represents a ring. inner_radius is expressed in the unit of the applicable length_quantity_type.
- outer_radius specifies the value of outer radius of an **mgm_disc**. outer_radius is expressed in the unit of the applicable length_quantity_type.
- start_angle specifies the value of the starting angle measured from the vector from p1 to p3 around the axis of revolution. start_angle defines the position of the **mgm_disc** segment. start_angle is expressed in the unit of the applicable plane_angle_quantity_type.
- end_angle specifies the value of the end angle measured from the vector from p1 to p3 around the axis of revolution. end_angle defines together with start_angle the size of the **mgm_disc** segment. end_angle is expressed in the unit of the applicable plane_angle_quantity_type.

Formal propositions:

- wr1: the coordinates of points p1, p2, p3 shall all use the length_quantity_type specified in the quantity_context of the containing model.
- wr2: inner_radius and outer_radius shall use the length_quantity_type specified in the quantity_context of the containing model.
- wr3: start_angle and end_angle shall use the plane_angle_quantity_type specified in the quantity_context of the containing model.
- wr4: points p1, p2 and p3 shall span an orthogonal system within the tolerance of the applicable point_coincidence_length uncertainty.
- wr5: the inner_radius shall be greater than or equal to zero.
- wr6: the outer_radius shall be greater than the inner_radius plus the applicable point_coincidence_length uncertainty.
- wr7: start_angle and end_angle shall comply with the following constraints: (1) both shall lie in the interval -360 to +360 degrees, (2) end_angle shall be greater than start_angle, (3) the difference between end_angle and start_angle shall be 360 degrees or less. These constraints shall be met while taking into account the applicable numerical tolerance.

4.3.6.14 ENTITY mgm_cylinder

An **mgm_cylinder** is a type of mgm_primitive_bounded_surface that defines a cylinder or cylinder segment surface.

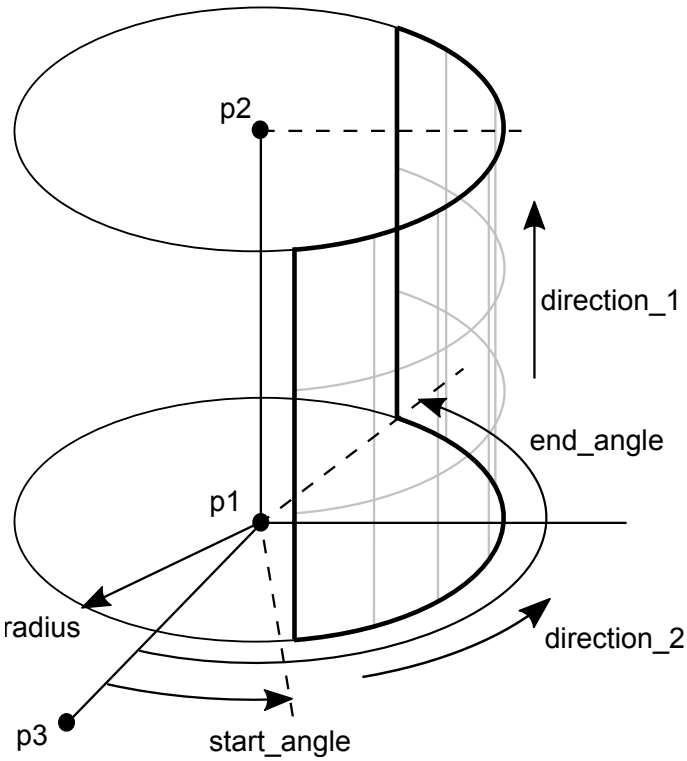

Figure 5 Sketch of an **mgm_cylinder**

The positive normal on the surface is defined outwards from the cylinder or cylinder-segment.

The first parametric direction (direction-1) is defined along the direction defined by the vector from p1 to p2. The second parametric direction (direction-2) is defined along the positive direction of revolution about the vector from p1 to p2.

The height of the **mgm_cylinder** is equal to the distance between p1 and p2.

Express specification:

```
ENTITY mgm_cylinder
  SUBTYPE OF(mgm_primitive_bounded_surface);
  p1          : mgm_3d_cartesian_point;
  p2          : mgm_3d_cartesian_point;
  p3          : mgm_3d_cartesian_point;
  radius      : nrf_real_quantity_value_prescription;
  start_angle : nrf_real_quantity_value_prescription;
  end_angle   : nrf_real_quantity_value_prescription;
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2, p3], geometric_item.containing_model);
  wr2: (radius.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'));
  wr3: (start_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'))
    AND (end_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'));
  wr4: mgm_verify_points_span_orthogonal_system(p1, p2, p3,
    mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
  wr5: radius.val > mgm_get_context_uncertainty_value(
    geometric_item.containing_model, 'point_coincidence_length');
```



```
  wr6: mgm_verify_start_and_end_angles(
    start_angle.val, end_angle.val, mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
END_ENTITY;
```

Attribute definitions:

- p1 specifies an mgm_3d_cartesian_point that is the centre of the sphere and defines together with p2 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2.
- p2 specifies an mgm_3d_cartesian_point that defines together with p1 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2. The height of the **mgm_cylinder** is equal to the distance between p1 and p2.
- p3 specifies an mgm_3d_cartesian_point that defines together with p1 and p2 the reference plane from which start_angle and end_angle are measured.
- radius specifies the radius of the cylinder.
- start_angle specifies the value of the starting angle measured from the vector from p1 to p3 around the axis of revolution. start_angle defines the position of the **mgm_cylinder** segment.
- end_angle specifies the value of the end angle measured from the vector from p1 to p3 around the axis of revolution. end_angle defines together with start_angle the size of the **mgm_cylinder** segment.

Formal propositions:

- wr1: the coordinates of points p1, p2, p3 shall all use the length_quantity_type specified in the quantity_context of the containing model.
- wr2: radius shall use the length_quantity_type specified in the quantity_context of the containing model.
- wr3: start_angle and end_angle shall use the plane_angle_quantity_type specified in the quantity_context of the containing model.
- wr4: points p1, p2 and p3 shall span an orthogonal system within the tolerance of the applicable point_coincidence_length uncertainty.
- wr5: radius shall be greater than the applicable point_coincidence_length uncertainty.
- wr6: start_angle and end_angle shall comply with the following constraints: (1) both shall lie in the interval -360 to +360 degrees, (2) end_angle shall be greater than start_angle, (3) the difference between end_angle and start_angle shall be 360 degrees or less. These constraints shall be met while taking into account the applicable numerical tolerance.

4.3.6.15 ENTITY mgm_cone

An **mgm_cone** is a type of mgm_primitive_bounded_surface that defines a cone or cone-segment surface.

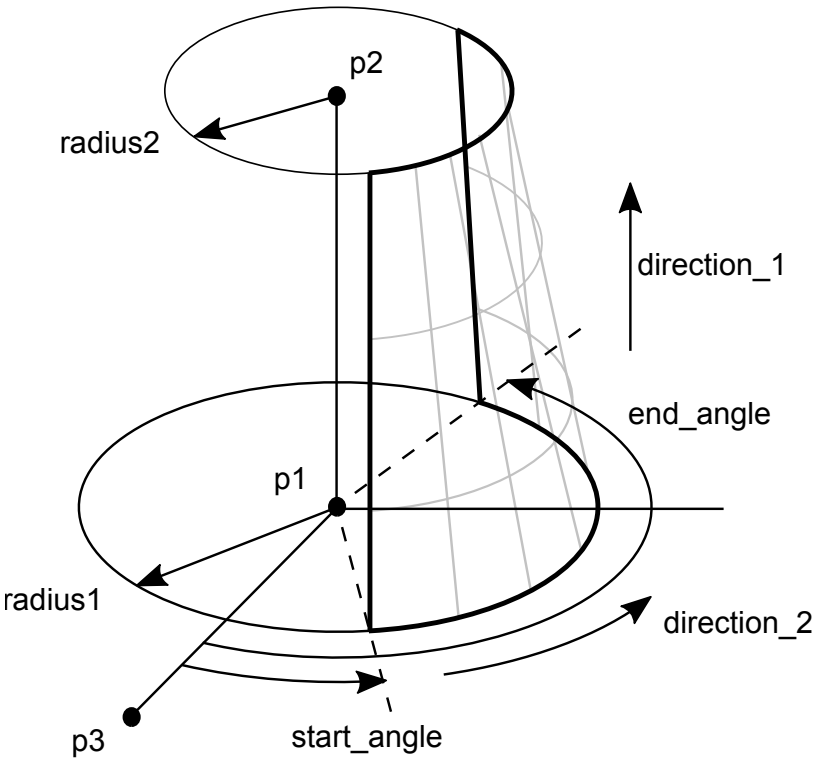

Figure 6 Sketch of an **mgm_cone**

The positive normal on the surface is defined outwards from the cone or cone-segment.

The first parametric direction (direction-1) is defined by the vector from p1 to p2. The second parametric direction (direction-2) is defined along the positive direction of revolution about the vector pointing from p1 to p2.

The height of the **mgm_cone** is equal to the distance between p1 and p2.

Express specification:

```
ENTITY mgm_cone
  SUBTYPE OF(mgm_primitive_bounded_surface);
  p1          : mgm_3d_cartesian_point;
  p2          : mgm_3d_cartesian_point;
  p3          : mgm_3d_cartesian_point;
  radius1     : nrf_real_quantity_value_prescription;
  radius2     : nrf_real_quantity_value_prescription;
  start_angle : nrf_real_quantity_value_prescription;
  end_angle   : nrf_real_quantity_value_prescription;
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2, p3], geometric_item.containing_model);
  wr2: (radius1.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'))
    AND (radius2.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'));
  wr3: (start_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'))
    AND (end_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'));
  wr4: mgm_verify_points_span_orthogonal_system(p1, p2, p3,
    mgm_get_context_uncertainty_value(
```



```
      geometric_item.containing_model, 'point_coincidence_length'));
  wr5: (radius1.val >= 0.0) AND (radius2.val >= 0.0);
  wr6: (radius1.val > mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'))
    OR (radius2.val > mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
  wr7: mgm_verify_start_and_end_angles(
    start_angle.val, end_angle.val, mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
END_ENTITY;
```

Attribute definitions:

- p1 specifies an mgm_3d_cartesian_point that is the centre of the sphere and defines together with p2 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2.
- p2 specifies an mgm_3d_cartesian_point that defines together with p1 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2. The height of the **mgm_cone** is equal to the distance between p1 and p2.
- p3 specifies an mgm_3d_cartesian_point that defines together with p1 and p2 the reference plane from which start_angle and end_angle are measured.
- radius1 specifies the radius of the cone at the plane in which p1 is located.
- radius2 specifies the radius of the cone at the plane in which p2 is located.
- start_angle specifies the value of the starting angle measured from the vector from p1 to p3 around the axis of revolution. start_angle defines the position of the **mgm_cone** segment.
- end_angle specifies the value of the end angle measured from the vector from p1 to p3 around the axis of revolution. end_angle defines together with start_angle the size of the **mgm_cone** segment.

Formal propositions:

- wr1: the coordinates of points p1, p2, p3 shall all use the length_quantity_type specified in the quantity_context of the containing model.
- wr2: radius shall use the length_quantity_type specified in the quantity_context of the containing model.
- wr3: start_angle and end_angle shall use the plane_angle_quantity_type specified in the quantity_context of the containing model.
- wr4: points p1, p2 and p3 shall span an orthogonal system within the tolerance of the applicable point_coincidence_length uncertainty.
- wr5: radius1 and radius2 shall both be greater than or equal to zero.
- wr6: At least one of radius1 and radius2 shall have a value greater than the applicable point_coincidence_length uncertainty.
- wr7: start_angle and end_angle shall comply with the following constraints: (1) both shall lie in the interval -360 to +360 degrees, (2) end_angle shall be greater than start_angle, (3) the difference between end_angle and start_angle shall be 360 degrees or less. These constraints shall be met while taking into account the applicable numerical tolerance.

4.3.6.16 ENTITY mgm_sphere

An **mgm_sphere** is a type of mgm_primitive_bounded_surface that defines a complete or partial spherical bounded surface.

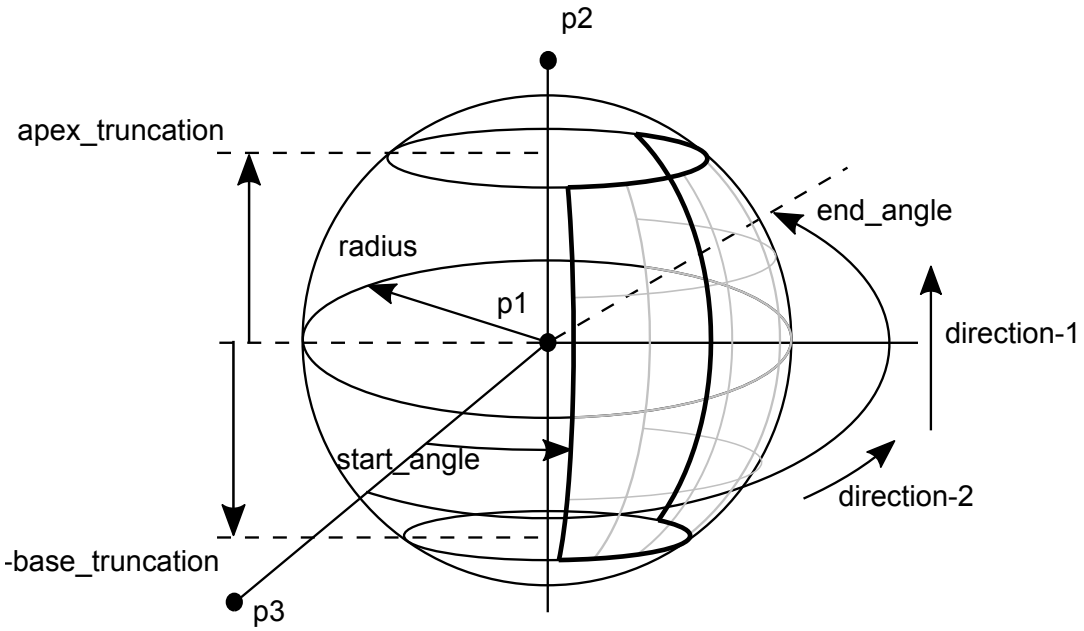

Figure 7 Sketch of an **mgm_sphere**

The positive normal on the surface is defined outwards from the sphere or sphere-segment.

The first parametric direction (direction-1) is defined by the vector from p1 to p2. The second parametric direction (direction-2) is defined along the positive direction of revolution about the vector pointing from p1 to p2.

Express specification:

```
ENTITY mgm_sphere
  SUBTYPE OF(mgm_primitive_bounded_surface);
  p1              : mgm_3d_cartesian_point;
  p2              : mgm_3d_cartesian_point;
  p3              : mgm_3d_cartesian_point;
  radius          : nrf_real_quantity_value_prescription;
  base_truncation : nrf_real_quantity_value_prescription;
  apex_truncation : nrf_real_quantity_value_prescription;
  start_angle     : nrf_real_quantity_value_prescription;
  end_angle       : nrf_real_quantity_value_prescription;
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2, p3], geometric_item.containing_model);
  wr2: (radius.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'))
    AND (base_truncation.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'))
    AND (apex_truncation.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'));
  wr3: (start_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'))
    AND (end_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'));
  wr4: mgm_verify_points_span_orthogonal_system(p1, p2, p3,
    mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
  wr5: radius.val >
 mgm_get_context_uncertainty_value(geometric_item.containing_model, 'point_coincidence_length');
  wr6: base_truncation.val >= -radius.val -
```



```
 mgm_get_context_uncertainty_value(geometric_item.containing_model, 'point_coincidence_length');
  wr7: apex_truncation.val <= radius.val +
 mgm_get_context_uncertainty_value(geometric_item.containing_model, 'point_coincidence_length');
  wr8: apex_truncation.val > base_truncation.val +
 mgm_get_context_uncertainty_value(geometric_item.containing_model, 'point_coincidence_length');
  wr9: mgm_verify_start_and_end_angles(
    start_angle.val, end_angle.val, mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
END_ENTITY;
```

Attribute definitions:

- p1 specifies an mgm_3d_cartesian_point that is the centre of the sphere and defines together with p2 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2.
- p2 specifies an mgm_3d_cartesian_point that defines together with p1 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2.
- p3 specifies an mgm_3d_cartesian_point that defines together with p1 and p2 the reference plane from which start_angle and end_angle are measured.
- radius specifies the radius of the **mgm_sphere**.
- base_truncation specifies specifies the position of base truncation plane measured from p1 along the axis of revolution. The base truncation plane is the plane perpendicular to the axis of revolution. If apex_truncation is set equal to radius and base_truncation is set equal to minus radius a complete sphere is defined.
- apex_truncation specifies the position of apex truncation plane measured from p1 along the axis of revolution. The apex truncation plane is the plane perpendicular to the axis of revolution. If apex_truncation is set equal to radius and base_truncation is set equal to minus radius a complete sphere is defined.
- start_angle specifies the angle at which the sphere segment starts. start_angle is measured from the plane defined by vectors p1 to p2 and p1 to p3 and around the axis of revolution.
- end_angle specifies the angle at which the sphere segment ends. end_angle is measured from the plane defined by vectors p1 to p2 and p1 to p3 and around the axis of revolution. end_angle defines together with start_angle the extent of the **mgm_sphere** segment. A complete sphere is defined when end_angle minus start_angle is equal to 360 degrees.

Formal propositions:

- wr1: the coordinates of points p1, p2, p3 shall all use the length_quantity_type specified in the quantity_context of the containing model.
- wr2: radius, base_truncation and apex_truncation shall use the length_quantity_type specified in the quantity_context of the containing model.
- wr3: start_angle and end_angle shall use the plane_angle_quantity_type specified in the quantity_context of the containing model.
- wr4: points p1, p2 and p3 shall span an orthogonal system within the tolerance of the applicable point_coincidence_length uncertainty.
- wr5: radius shall be greater than the applicable point_coincidence_length uncertainty.
- wr6: base_truncation shall be greater than or equal to the negative radius minus the applicable point_coincidence_length uncertainty.
- wr7: apex_truncation shall be less than or equal to the radius plus the applicable point_coincidence_length uncertainty.

— wr8: apex_truncation shall be greater than base_truncation plus the applicable point_coincidence_length uncertainty.

— wr9: start_angle and end_angle shall comply with the following constraints: (1) both shall lie in the interval -360 to +360 degrees, (2) end_angle shall be greater than start_angle, (3) the difference between end_angle and start_angle shall be 360 degrees or less. These constraints shall be met while taking into account the applicable numerical tolerance.

4.3.6.17 ENTITY mgm_paraboloid

An **mgm_paraboloid** is a type of mgm_primitive_bounded_surface that defines a complete or partial paraboloid surface.

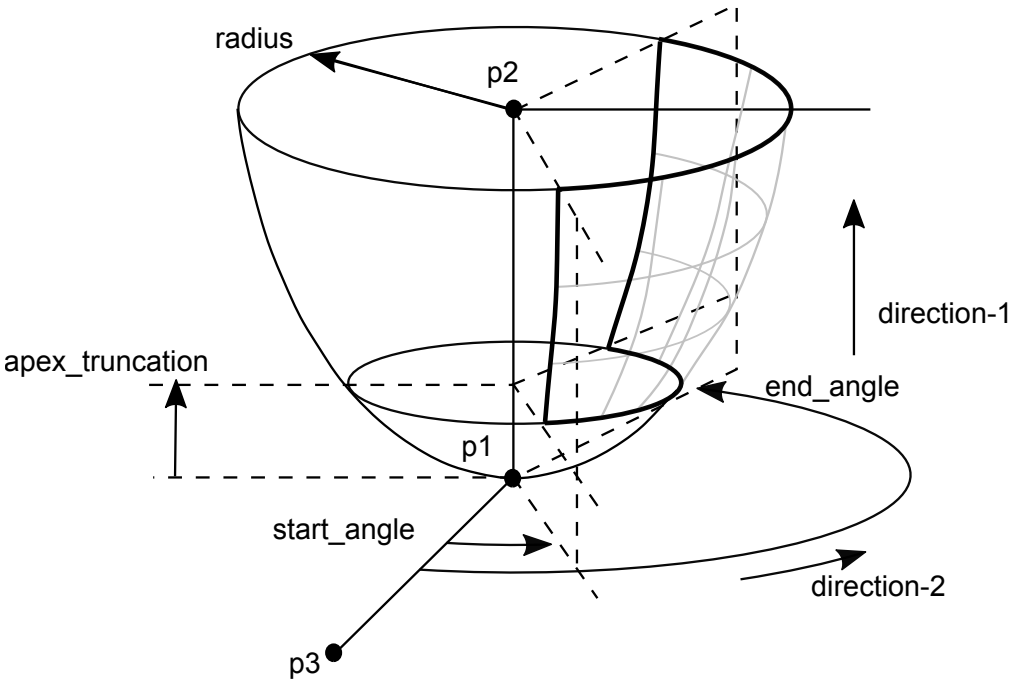

Figure 8 Sketch of an **mgm_paraboloid**

The positive normal on the surface is defined outwards from the paraboloid or paraboloid-segment.

The first parametric direction (direction-1) is defined by the vector from p1 to p2. The second parametric direction (direction-2) is defined along the positive direction of revolution about the vector pointing from p1 to p2.

The height of the **mgm_paraboloid** is equal to the distance between p1 and p2.

Express specification:

```
ENTITY mgm_paraboloid
  SUBTYPE OF(mgm_primitive_bounded_surface);
  p1              : mgm_3d_cartesian_point;
  p2              : mgm_3d_cartesian_point;
  p3              : mgm_3d_cartesian_point;
  radius          : nrf_real_quantity_value_prescription;
  apex_truncation : nrf_real_quantity_value_prescription;
  start_angle     : nrf_real_quantity_value_prescription;
  end_angle       : nrf_real_quantity_value_prescription;
```



```
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2, p3], geometric_item.containing_model);
  wr2: (radius.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'))
    AND (apex_truncation.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'));
  wr3: (start_angle.quantity_type :=:
   mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'))
    AND (end_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'));
  wr4: mgm_verify_points_span_orthogonal_system(p1, p2, p3,
    mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
  wr5: radius.val >
 mgm_get_context_uncertainty_value(geometric_item.containing_model, 'point_coincidence_length');
  wr6: apex_truncation.val >= 0.0;
  wr7: mgm_compute_distance_between_points(p1, p2) > apex_truncation.val +
 mgm_get_context_uncertainty_value(geometric_item.containing_model, 'point_coincidence_length');
  wr8: mgm_verify_start_and_end_angles(
    start_angle.val, end_angle.val, mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
END_ENTITY;
```

Attribute definitions:

- p1 specifies an mgm_3d_cartesian_point that defines the location of the apex of the paraboloid and that defines together with p2 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2.
- p2 specifies an mgm_3d_cartesian_point that defines together with p1 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2. Point p2 also defines the base truncation plane of the paraboloid. The base truncation plane is perpendicular to the axis of revolution. The height of the paraboloid is equal to the distance between p1 and p2.
- p3 specifies an mgm_3d_cartesian_point that defines together with p1 and p2 the reference plane from which start_angle and end_angle are measured.
- radius specifies the radius of the paraboloid at the base truncation plane defined by p2.
- apex_truncation specifies the position of apex truncation plane measured from p1 along the axis of revolution. The apex truncation plane is perpendicular to the axis of revolution.
- start_angle specifies the angle at which the paraboloid segment starts. start_angle is measured from the plane defined by vectors p1 to p2 and p1 to p3 and around the axis of revolution.
- end_angle specifies the angle at which the paraboloid segment ends. end_angle is measured from the plane defined by vectors p1 to p2 and p1 to p3 and around the axis of revolution. end_angle defines together with start_angle the extent of the **mgm_paraboloid** segment. A complete paraboloid is defined when end_angle minus start_angle is equal to 360 degrees.

Formal propositions:

- wr1: the coordinates of points p1, p2, p3 shall all use the length_quantity_type specified in the quantity_context of the containing model.
- wr2: radius and apex_truncation shall use the length_quantity_type specified in the quantity_context of the containing model.
- wr3: start_angle and end_angle shall use the plane_angle_quantity_type specified in the quantity_context of the containing model.

- wr4: points p1, p2 and p3 shall span an orthogonal system within the tolerance of the applicable point_coincidence_length uncertainty.
- wr5: radius shall be greater than the applicable point_coincidence_length uncertainty.
- wr6: apex_truncation shall be greater than or equal to zero.
- wr7: the height of the paraboloid (that is the distance between p1 and p2) shall be greater than the apex_truncation plus the applicable point_coincidence_length uncertainty.
- wr8: start_angle and end_angle shall comply with the following constraints: (1) both shall lie in the interval -360 to +360 degrees, (2) end_angle shall be greater than start_angle, (3) the difference between end_angle and start_angle shall be 360 degrees or less. These constraints shall be met while taking into account the applicable numerical tolerance.

4.3.6.18 ENTITY mgm_primitive_solid

An **mgm_primitive_solid** is an abstract supertype that specifies a simple regular solid shape in 3D space. It is a generic object that specifies the attributes common to all primitive solids.

Express specification:

```
ENTITY mgm_primitive_solid
  ABSTRACT SUPERTYPE OF(ONEOF(
    mgm_infinite_solid_by_plane,
    mgm_infinite_solid_cylinder,
    mgm_solid_cylinder,
    mgm_solid_cone,
    mgm_solid_sphere,
    mgm_solid_paraboloid,
    mgm_solid_box,
    mgm_solid_triangular_prism));
  geometric_item      : mgm_any_meshed_geometric_item;
END_ENTITY;
```

Attribute definitions:

- geometric_item specifies the mgm_any_meshed_geometric_item that is using this **mgm_primitive_solid** to define its solid shape.

4.3.6.19 ENTITY mgm_infinite_solid_by_plane

An **mgm_infinite_solid_by_plane** is a type of mgm_primitive_solid that specifies an infinitely extending solid shape on one side of a plane. The plane is defined by two points. The plane goes through point p1 and the positive normal on the plane is spanned by the vector from point p1 to point p2. This solid is typically used to define a half space that is delimited by an infinite plane for use in boolean geometric construction operations.

For the purpose of applying an mgm_half_space_selector_type the OUTSIDE of the solid is defined as all points on the side of the positive normal on the plane and consequently the INSIDE is defined as all points on the side of the negative normal.

Express specification:

```
ENTITY mgm_infinite_solid_by_plane
  SUBTYPE OF(mgm_primitive_solid);
  p1     : mgm_3d_cartesian_point;
```



```
  p2     : mgm_3d_cartesian_point;
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2], geometric_item.containing_model);
  wr2: mgm_verify_no_coincident_points([p1, p2], mgm_get_context_uncertainty_value(
    geometric_item.containing_model, 'point_coincidence_length'));
END_ENTITY;
```

Formal propositions:

- wr1: the coordinates of points p1 and p2 shall use the length_quantity_type specified in the global_quantity_context of the containing model.
- wr2: the distance between points p1 and p2 shall be greater than the point_coincidence_length uncertainty.

4.3.6.20 ENTITY mgm_infinite_solid_cylinder

An **mgm_infinite_solid_cylinder** is a type of mgm_primitive_solid that specifies an infinitely long cylindrical solid shape. The axis of revolution of the cylindrical solid is defined by two points. This solid is typically used to define a cylindrical half space for use in boolean geometric construction operations.

Express specification:

```
ENTITY mgm_infinite_solid_cylinder
  SUBTYPE OF(mgm_primitive_solid);
  p1     : mgm_3d_cartesian_point;
  p2     : mgm_3d_cartesian_point;
  radius : nrf_real_quantity_value_prescription;
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2], geometric_item.containing_model);
  wr2: mgm_verify_no_coincident_points([p1, p2], mgm_get_context_uncertainty_value(
    geometric_item.containing_model, 'point_coincidence_length'));
  wr3: (radius.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'));
  wr4: radius.val > mgm_get_context_uncertainty_value(
    geometric_item.containing_model, 'point_coincidence_length');
END_ENTITY;
```

Attribute definitions:

- p1 specifies the first point of the axis of revolution of the cylinder
- p2 specifies the second point of the axis of revolution of the cylinder
- radius specifies the radius of the cylinder.

Formal propositions:

- wr1: the coordinates of points p1 and p2 shall use the length_quantity_type specified in the global_quantity_context of the containing model.
- wr2: the distance between points p1 and p2 shall be greater than the point_coincidence_length uncertainty.
- wr3: radius shall use the length_quantity_type specified in the quantity_context of the containing model.
- wr4: radius shall be greater than the applicable point_coincidence_length uncertainty.

4.3.6.21 ENTITY mgm_solid_cylinder

An **mgm_solid_cylinder** is a type of mgm_primitive_solid that defines a solid cylinder or a solid cylinder segment. The cylinder segment is the part of the cylinder enclosed by the planes at start_angle and end_angle together with the planes normal to the axis at the points p1 and p2.

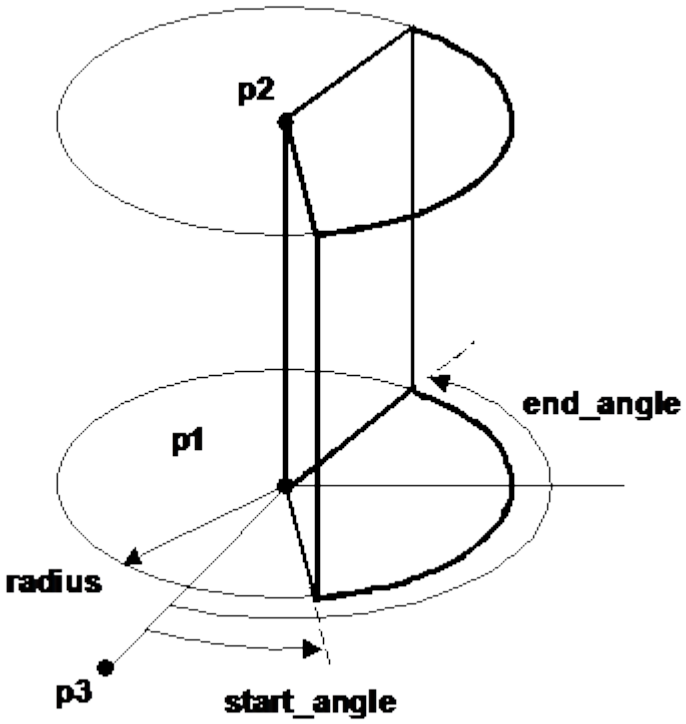

Figure 9 Sketch of an **mgm_solid_cylinder**

The height of the **mgm_solid_cylinder** is equal to the distance between p1 and p2.

Express specification:

```
ENTITY mgm_solid_cylinder
  SUBTYPE OF(mgm_primitive_solid);
  p1          : mgm_3d_cartesian_point;
  p2          : mgm_3d_cartesian_point;
  p3          : mgm_3d_cartesian_point;
  radius      : nrf_real_quantity_value_prescription;
  start_angle : nrf_real_quantity_value_prescription;
  end_angle   : nrf_real_quantity_value_prescription;
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2, p3], geometric_item.containing_model);
  wr2: (radius.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'));
  wr3: (start_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'))
    AND (end_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'));
  wr4: mgm_verify_points_span_orthogonal_system(p1, p2, p3,
    mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
  wr5: radius.val > mgm_get_context_uncertainty_value(
    geometric_item.containing_model, 'point_coincidence_length');
```



```
  wr6: mgm_verify_start_and_end_angles(
    start_angle.val, end_angle.val, mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
END_ENTITY;
```

Attribute definitions:

- p1 specifies an mgm_3d_cartesian_point that is is the centre of the cylinder and defines together with p2 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2.
- p2 specifies an mgm_3d_cartesian_point that is defines together with p1 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2. The height of the mgm_cylinder is equal to the distance between p1 and p2.
- p3 specifies an mgm_3d_cartesian_point that defines together with p1 and p2 the reference plane from which start_angle and end_angle are measured.
- radius specifies the radius of the cylinder.
- start_angle specifies the value of the starting angle measured from the vector from p1 to p3 around the axis of revolution. start_angle defines the position of the mgm_cylinder segment.
- end_angle specifies the value of the end angle measured from the vector from p1 to p3 around the axis of revolution. end_angle defines together with start_angle the size of the mgm_cylinder segment.

Formal propositions:

- wr1: the coordinates of points p1, p2, p3 shall all use the length_quantity_type specified in the quantity_context of the containing model.
- wr2: radius shall use the length_quantity_type specified in the quantity_context of the containing model.
- wr3: start_angle and end_angle shall use the plane_angle_quantity_type specified in the quantity_context of the containing model.
- wr4: points p1, p2 and p3 shall span an orthogonal system within the tolerance of the applicable point_coincidence_length uncertainty.
- wr5: radius shall be greater than the applicable point_coincidence_length uncertainty.
- wr6: start_angle and end_angle shall comply with the following constraints: (1) both shall lie in the interval -360 to +360 degrees, (2) end_angle shall be greater than start_angle, (3) the difference between end_angle and start_angle shall be 360 degrees or less. These constraints shall be met while taking into account the applicable numerical tolerance.

4.3.6.22 ENTITY mgm_solid_cone

An **mgm_solid_cone** is a type of mgm_primitive_solid that defines a solid cone or a solid cone segment. The cone segment is the part of the cone enclosed by the planes at start_angle and end_angle together with the planes normal to the axis at the points p1 and p2.

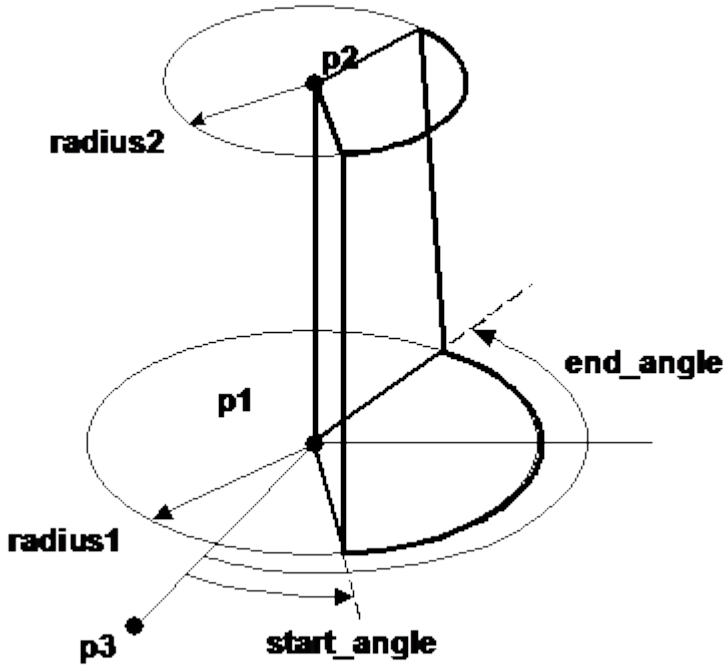

Figure 10 Sketch of an **mgm_solid_cone**

The height of the mgm_cone is equal to the distance between p1 and p2.

Express specification:

```
ENTITY mgm_solid_cone
  SUBTYPE OF(mgm_primitive_solid);
  p1          : mgm_3d_cartesian_point;
  p2          : mgm_3d_cartesian_point;
  p3          : mgm_3d_cartesian_point;
  radius1     : nrf_real_quantity_value_prescription;
  radius2     : nrf_real_quantity_value_prescription;
  start_angle : nrf_real_quantity_value_prescription;
  end_angle   : nrf_real_quantity_value_prescription;
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2, p3], geometric_item.containing_model);
  wr2: (radius1.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'))
    AND (radius2.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'));
  wr3: (start_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'))
    AND (end_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'));
  wr4: mgm_verify_points_span_orthogonal_system(p1, p2, p3,
    mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
  wr5: (radius1.val >= 0.0) AND (radius2.val >= 0.0);
  wr6: (radius1.val > mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'))
    OR (radius2.val > mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
  wr7: mgm_verify_start_and_end_angles(
    start_angle.val, end_angle.val, mgm_get_context_uncertainty_value(
```



```
      geometric_item.containing_model, 'point_coincidence_length'));
END_ENTITY;
```

Attribute definitions:

- p1 specifies an mgm_3d_cartesian_point that is is the centre of the cone and defines together with p2 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2.
- p2 specifies an mgm_3d_cartesian_point that is defines together with p1 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2. The height of the mgm_cone is equal to the distance between p1 and p2.
- p3 specifies an mgm_3d_cartesian_point that defines together with p1 and p2 the reference plane from which start_angle and end_angle are measured.
- radius1 specifies the radius of the cone at the plane in which p1 is located.
- radius2 specifies the radius of the cone at the plane in which p2 is located.
- start_angle specifies the value of the starting angle measured from the vector from p1 to p3 around the axis of revolution. start_angle defines the position of the mgm_cone segment.
- end_angle specifies the value of the end angle measured from the vector from p1 to p3 around the axis of revolution. end_angle defines together with start_angle the size of the mgm_cone segment.

Formal propositions:

- wr1: the coordinates of points p1, p2, p3 shall all use the length_quantity_type specified in the quantity_context of the containing model.
- wr2: radius shall use the length_quantity_type specified in the quantity_context of the containing model.
- wr3: start_angle and end_angle shall use the plane_angle_quantity_type specified in the quantity_context of the containing model.
- wr4: points p1, p2 and p3 shall span an orthogonal system within the tolerance of the applicable point_coincidence_length uncertainty.
- wr5: radius1 and radius2 shall both be greater than zero.
- wr6: At least one of radius1 and radius2 shall have a value greater than the applicable point_coincidence_length uncertainty.
- wr7: start_angle and end_angle shall comply with the following constraints: (1) both shall lie in the interval -360 to +360 degrees, (2) end_angle shall be greater than start_angle, (3) the difference between end_angle and start_angle shall be 360 degrees or less. These constraints shall be met while taking into account the applicable numerical tolerance.

4.3.6.23 ENTITY mgm_solid_sphere

An **mgm_solid_sphere** is a type of mgm_primitive_solid that defines a solid sphere or a solid sphere segment. The sphere segment is the part of the sphere enclosed by the planes at start_angle and end_angle, together with the planes at the apex truncation and base truncation.

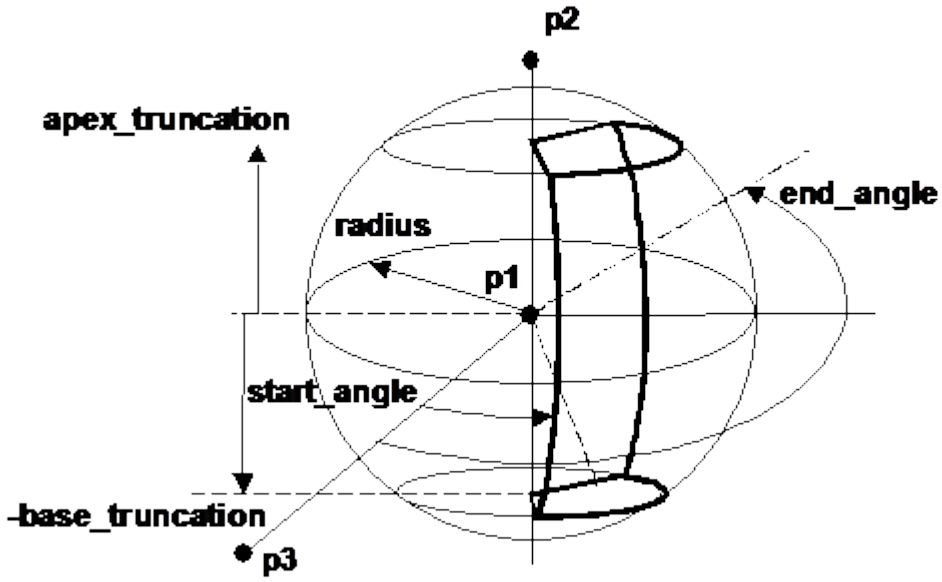

Figure 11 Sketch of an **mgm_solid_sphere**

Express specification:

```
ENTITY mgm_solid_sphere
  SUBTYPE OF(mgm_primitive_solid);
  p1               : mgm_3d_cartesian_point;
  p2               : mgm_3d_cartesian_point;
  p3               : mgm_3d_cartesian_point;
  radius           : nrf_real_quantity_value_prescription;
  base_truncation  : nrf_real_quantity_value_prescription;
  apex_truncation  : nrf_real_quantity_value_prescription;
  start_angle      : nrf_real_quantity_value_prescription;
  end_angle        : nrf_real_quantity_value_prescription;
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2, p3], geometric_item.containing_model);
  wr2: (radius.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'))
    AND (base_truncation.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'))
    AND (apex_truncation.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'));
  wr3: (start_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'))
    AND (end_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'));
  wr4: mgm_verify_points_span_orthogonal_system(p1, p2, p3,
    mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
  wr5: radius.val >
 mgm_get_context_uncertainty_value(geometric_item.containing_model, 'point_coincidence_length');
  wr6: base_truncation.val >= -radius.val -
 mgm_get_context_uncertainty_value(geometric_item.containing_model, 'point_coincidence_length');
  wr7: apex_truncation.val <= radius.val +
 mgm_get_context_uncertainty_value(geometric_item.containing_model, 'point_coincidence_length');
  wr8: apex_truncation.val > base_truncation.val +
 mgm_get_context_uncertainty_value(geometric_item.containing_model, 'point_coincidence_length');
  wr9: mgm_verify_start_and_end_angles(
    start_angle.val, end_angle.val, mgm_get_context_uncertainty_value(
```



```
      geometric_item.containing_model, 'point_coincidence_length'));
END_ENTITY;
```

Attribute definitions:

- p1 specifies an mgm_3d_cartesian_point that is is the centre of the sphere and defines together with p2 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2.
- p2 specifies an mgm_3d_cartesian_point that defines together with p1 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2.
- p3 specifies an mgm_3d_cartesian_point that defines together with p1 and p2 the reference plane from which start_angle and end_angle are measured.
- radius specifies the radius of the mgm_sphere.
- base_truncation specifies specifies the position of base truncation plane measured from p1 along the axis of revolution. The base truncation plane is the plane perpendicular to the axis of revolution. If apex_truncation is set equal to radius and base_truncation is set equal to minus radius a complete sphere is defined.
- apex_truncation specifies the position of apex truncation plane measured from p1 along the axis of revolution. The apex truncation plane is the plane perpendicular to the axis of revolution. If apex_truncation is set equal to radius and base_truncation is set to minus radius a complete sphere is defined.
- start_angle specifies the angle at which the sphere segment starts. start_angle is measured from the plane defined by vectors p1 to p2 and p1 to p3 and around the axis of revolution.
- end_angle specifies the angle at which the sphere segment ends. end_angle is measured from the plane defined by vectors p1 to p2 and p1 to p3 and around the axis of revolution. end_angle defines together with start_angle the extent of the mgm_sphere segment. A complete sphere is defined when end_angle minus start_angle is equal to 360 degrees.

Formal propositions:

- wr1: the coordinates of points p1, p2, p3 shall all use the length_quantity_type specified in the quantity_context of the containing model.
- wr2: radius shall use the length_quantity_type specified in the quantity_context of the containing model.
- wr3: start_angle and end_angle shall use the plane_angle_quantity_type specified in the quantity_context of the containing model.
- wr4: points p1, p2 and p3 shall span an orthogonal system within the tolerance of the applicable point_coincidence_length uncertainty.
- wr5: radius shall be greater than the applicable point_coincidence_length uncertainty.
- wr6: base_truncation shall be greater than or equal to the negative radius minus the applicable point_coincidence_length uncertainty.
- wr7: apex_truncation shall be less than or equal to the radius plus the applicable point_coincidence_length uncertainty.
- wr8: apex_truncation shall be greater than base_truncation plus the applicable point_coincidence_length uncertainty.
- wr9: start_angle and end_angle shall comply with the following constraints: (1) both shall lie in the interval -360 to +360 degrees, (2) end_angle shall be greater than start_angle, (3) the difference

between end_angle and start_angle shall be 360 degrees or less. These constraints shall be met while taking into account the applicable numerical tolerance.

4.3.6.24 ENTITY mgm_solid_paraboloid

An **mgm_solid_paraboloid** is a type of mgm_primitive_solid that defines a solid paraboloid or a solid paraboloid segment. The paraboloid segment is the part of the paraboloid enclosed by the planes at start_angle and end_angle, together with the planes at point p2 and the base truncation.

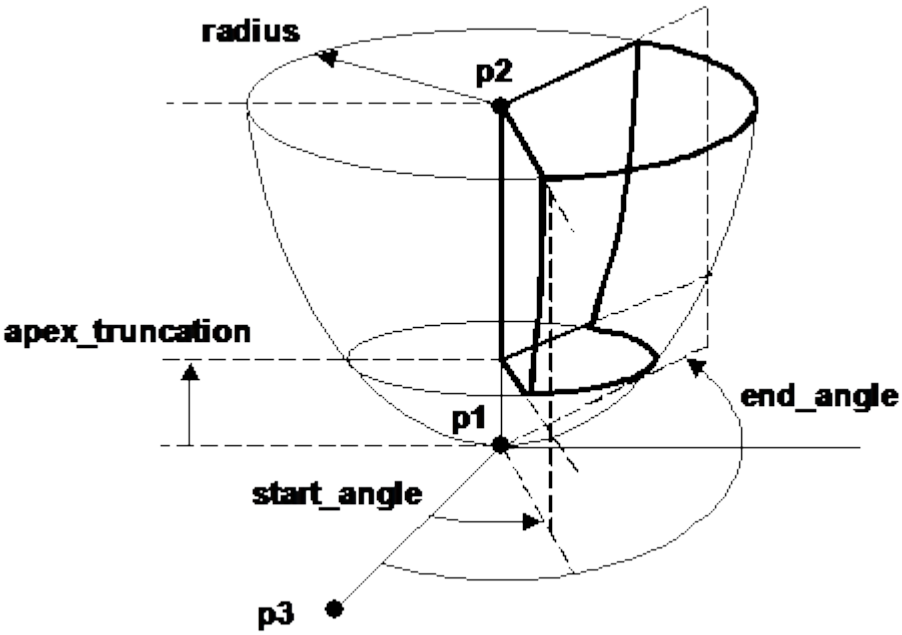

Figure 12 Sketch of an **mgm_solid_paraboloid**

The height of the mgm_paraboloid is equal to the distance between p1 and p2.

Express specification:

```
ENTITY mgm_solid_paraboloid
  SUBTYPE OF(mgm_primitive_solid);
  p1              : mgm_3d_cartesian_point;
  p2              : mgm_3d_cartesian_point;
  p3              : mgm_3d_cartesian_point;
  radius          : nrf_real_quantity_value_prescription;
  apex_truncation : nrf_real_quantity_value_prescription;
  start_angle     : nrf_real_quantity_value_prescription;
  end_angle       : nrf_real_quantity_value_prescription;
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2, p3], geometric_item.containing_model);
  wr2: (radius.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'))
    AND (apex_truncation.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'length'));
  wr3: (start_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'))
    AND (end_angle.quantity_type :=:
 mgm_get_context_quantity_type(geometric_item.containing_model, 'plane_angle'));
  wr4: mgm_verify_points_span_orthogonal_system(p1, p2, p3,
```



```
    mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
  wr5: radius.val >
mgm_get_context_uncertainty_value(geometric_item.containing_model, 'point_coincidence_length');
  wr6: apex_truncation.val >= 0.0;
  wr7: mgm_compute_distance_between_points(p1, p2) > apex_truncation.val +
mgm_get_context_uncertainty_value(geometric_item.containing_model, 'point_coincidence_length');
  wr8: mgm_verify_start_and_end_angles(
    start_angle.val, end_angle.val, mgm_get_context_uncertainty_value(
      geometric_item.containing_model, 'point_coincidence_length'));
END_ENTITY;
```

Attribute definitions:

- p1 specifies an mgm_3d_cartesian_point that defines the location of the apex of the paraboloid and that defines together with p2 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2.
- p2 specifies an mgm_3d_cartesian_point that defines together with p1 the axis of revolution. The positive direction of the axis of revolution is in the direction from p1 to p2. Point p2 also defines the base truncation plane of the paraboloid. The base truncation plane is perpendicular to the axis of revolution. The height of the paraboloid is equal to the distance between p1 and p2.
- p3 specifies an mgm_3d_cartesian_point that defines together with p1 and p2 the reference plane from which start_angle and end_angle are measured.
- radius specifies the radius of the paraboloid at the base truncation plane defined by p2.
- apex_truncation specifies the position of apex truncation plane measured from p1 along the axis of revolution. The apex truncation plane is perpendicular to the axis of revolution.
- start_angle specifies the angle at which the paraboloid segment starts. start_angle is measured from the plane defined by vectors p1 to p2 and p1 to p3 and around the axis of revolution.
- end_angle specifies the angle at which the paraboloid segment ends. end_angle is measured from the plane defined by vectors p1 to p2 and p1 to p3 and around the axis of revolution. end_angle defines together with start_angle the extent of the mgm_paraboloid segment. A complete paraboloid is defined when end_angle minus start_angle is equal to 360 degrees.

Formal propositions:

- wr1: the coordinates of points p1, p2, p3 shall all use the length_quantity_type specified in the quantity_context of the containing model.
- wr2: radius shall use the length_quantity_type specified in the quantity_context of the containing model.
- wr3: start_angle and end_angle shall use the plane_angle_quantity_type specified in the quantity_context of the containing model.
- wr4: points p1, p2 and p3 shall span an orthogonal system within the tolerance of the applicable point_coincidence_length uncertainty.
- wr5: radius shall be greater than the applicable point_coincidence_length uncertainty.
- wr6: apex_truncation shall be greater than or equal to zero.
- wr7: the height of the paraboloid (that is the distance between p1 and p2) shall be greater than the apex_truncation plus the applicable point_coincidence_length uncertainty.
- wr8: start_angle and end_angle shall comply with the following constraints: (1) both shall lie in the interval -360 to +360 degrees, (2) end_angle shall be greater than start_angle, (3) the difference

between end_angle and start_angle shall be 360 degrees or less. These constraints shall be met while taking into account the applicable numerical tolerance.

4.3.6.25 ENTITY mgm_solid_box

An **mgm_solid_box** is a type of mgm_primitive_solid that defines a rectangular box through four points p1, p2, p3 and p4. The points define four vertices of the box as illustrated in the figure below. The other four vertices of the box are implied to be: p2 plus vector p1 to p3, p2 plus vector p1 to p4, p3 plus vector p1 to p4, and p2 plus vector p1 to p3 plus vector p1 to p4.

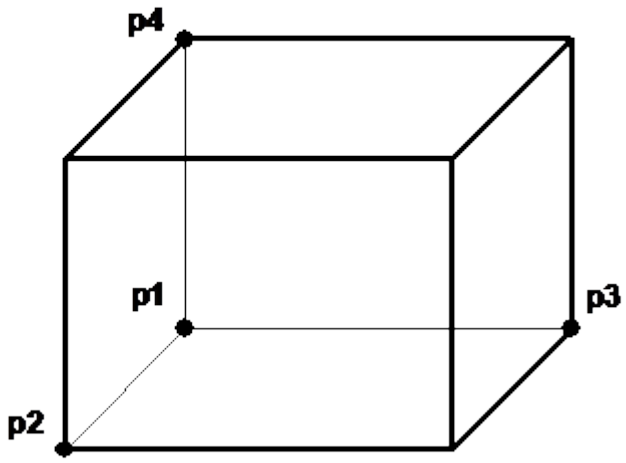

Figure 13 Sketch of an **mgm_solid_box**

Express specification:

```
ENTITY mgm_solid_box
  SUBTYPE OF(mgm_primitive_solid);
  p1 : mgm_3d_cartesian_point;
  p2 : mgm_3d_cartesian_point;
  p3 : mgm_3d_cartesian_point;
  p4 : mgm_3d_cartesian_point;
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2, p3, p4], geometric_item.containing_model);
  wr2: mgm_verify_solid_box(SELF);
END_ENTITY;
```

Attribute definitions:

- p1 specifies the first vertex of the solid box
- p2 specifies the second vertex of the solid box
- p3 specifies the third vertex of the solid box
- p4 specifies the fourth vertex of the solid box

Formal propositions:

— wr1: the coordinates of points p1, p2, p3, p4 shall all use the length_quantity_type specified in the quantity_context of the containing model.

— wr2: the specification shall meet the following criteria: (1) the points p1, p2, p3, p4 do not coincide, (2) the points p1, p2, p3 span an orthogonal system, (3) vector p1 to p4 has the same direction as the positive normal on the plane spanned by vector p1 to p2 and vector p1 to p3.

4.3.6.26 ENTITY mgm_solid_triangular_prism

An **mgm_solid_triangular_prism** is a type of mgm_primitive_solid that defines a triangular prism through a triangle defined by three points and a height defined by a fourth point.

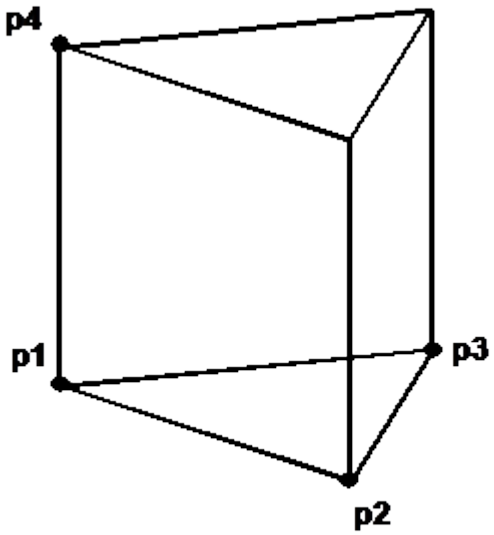

Figure 14 Sketch of an **mgm_solid_triangular_prism**

Express specification:

```
ENTITY mgm_solid_triangular_prism
  SUBTYPE OF(mgm_primitive_solid);
  p1 : mgm_3d_cartesian_point;
  p2 : mgm_3d_cartesian_point;
  p3 : mgm_3d_cartesian_point;
  p4 : mgm_3d_cartesian_point;
WHERE
  wr1: mgm_verify_points_use_context_length_quantity_type(
    [p1, p2, p3, p4], geometric_item.containing_model);
  wr2: mgm_verify_solid_triangular_prism(SELF);
END_ENTITY;
```

Attribute definitions:

— p1 specifies an mgm_3d_cartesian_point that defines the first vertex of a triangular base of the triangular prism

— p2 specifies an mgm_3d_cartesian_point that defines the second vertex of a triangular base of the triangular prism

— p3 specifies an mgm_3d_cartesian_point that defines the third vertex of a triangular base of the triangular prism

— p4 specifies an mgm_3d_cartesian_point that defines the height of the triangular prism

Formal propositions:

— wr1: the coordinates of points p1, p2, p3, p4 shall all use the length_quantity_type specified in the quantity_context of the containing model.

— wr2: the specification shall meet the following criteria: (1) the points p1, p2, p3 do not coincide, (2) the points p1, p2, p3 are not colinear, (3) vector p1 to p4 has the same direction as the positive normal on the plane spanned by vector p1 to p2 and vector p1 to p3.

4.3.6.27 ENTITY mgm_qualified_compound_meshed_primitive_bounded_surface

An **mgm_qualified_compound_meshed_primitive_bounded_surface** is a type of mgm_compound_meshed_geometric_item that collects mgm_meshed_primitive_bounded_surface instances to a higher order meshed primitive shape. The kind of shape is designated by the qualifier attribute.

EXAMPLE 1 - A box can be represented by an **mgm_qualified_compound_meshed_primitive_bounded_surface** consisting of six mgm_meshed_primitive_bounded_surface instances using mgm_rectangle surfaces. The value of the qualifier could then be "closed_six_sided_box".

Express specification:

```
ENTITY mgm_qualified_compound_meshed_primitive_bounded_surface
  SUBTYPE OF(mgm_compound_meshed_geometric_item);
 SELF\mgm_compound_meshed_geometric_item.geometric_items :
    LIST [1:?] OF UNIQUE mgm_meshed_primitive_bounded_surface;
  qualifier : nrf_enumeration_quantity_value_literal;
WHERE
  wr1: qualifier.quantity_type.name = 'compound_surface_qualifier';
END_ENTITY;
```

Attribute definitions:

— geometric_items is redeclared to specify the list of mgm_meshed_primitive_bounded_surface instances that form a higher order primitive.

— qualifier specifies the designation of the kind of higher order shape represented by an **mgm_qualified_compound_meshed_primitive_bounded_surface**.

Formal propositions:

— wr1: The name of the quantity_type of the qualifier shall be 'compound_surface_qualifier'.

4.3.6.28 ENTITY mgm_face

An **mgm_face** represents a face meshed on the geometry of an mgm_meshed_primitive_bounded_surface. For reasons of efficiency no id, name and description attributes are included.

Express specification:

```
ENTITY mgm_face
  SUBTYPE OF(nrf_observable_item);
```



```
  corresponding_node : OPTIONAL nrf_network_node;
END_ENTITY;
```

Attribute definitions:

— corresponding_node optionally specifies the nrf_network_node that corresponds to an **mgm_face**. The corresponding_node is not specified when no other nrf_network_model is associated with an mgm_meshed_geometric_model.

4.3.6.29 ENTITY mgm_face_pair

An **mgm_face_pair** is a type of nrf_observable_item_relationship that specifies a pair of mgm_face instances.

EXAMPLE 2 - A radiative exchange factor quantity type can be associated with a pair of thermal radiative faces that is represented by an **mgm_face_pair**.

Express specification:

```
ENTITY mgm_face_pair
  SUBTYPE OF(nrf_observable_item_relationship);
  SELF\nrf_observable_item_relationship.items : LIST [2:2] OF mgm_face;
END_ENTITY;
```

Attribute definitions:

— items specifies the pair of mgm_face instances.

4.3.6.30 ENTITY mgm_enclosure

An **mgm_enclosure** is a named grouping of mgm_meshed_primitive_bounded_surface instances.

EXAMPLE 3 - An **mgm_enclosure** can be used for the purpose of efficient thermal radiative computations. The faces in one enclosure cannot exchange heat through thermal radiation with faces in another enclosure.

Express specification:

```
ENTITY mgm_enclosure;
  id           : nrf_identifier;
  name         : nrf_label;
  description  : nrf_text;
  surfaces     : LIST [1:?] OF mgm_meshed_primitive_bounded_surface;
  active_sides : LIST [1:?] OF mgm_active_side_type;
INVERSE
  containing_model : mgm_meshed_geometric_model FOR enclosures;
UNIQUE
  ur1: containing_model, id;
-- WHERE
--   wr1: mgm_verify_enclosure_faces(SELF);
END_ENTITY;
```

Attribute definitions:

— id specifies the identifier of an **mgm_enclosure**.

— name specifies the human-interpretable name of an **mgm_enclosure**.

- description specifies the textual description for an **mgm_enclosure**.
- surfaces specifies the list of mgm_meshed_primitive_bounded_surface instances that belong to the enclosure.
- active_sides specifies for each surface in surfaces which side is part of the enclosure.
- containing_model specifies the mgm_meshed_geometric_model containing an **mgm_enclosure**.

Formal propositions:

- ur1: the combination of containing_model and id must be unique within the dataset.
- wr1: all surfaces shall be part of the containing_model, either directly or through a submodel and the active side as specified in active_sides shall be one or both of the active sides specified for the corresponding surface.

4.3.6.31 FUNCTION mgm_verify_transformation

The function **mgm_verify_transformation** verifies that all elements in the transformation of an mgm_any_meshed_geometric_item use the applicable context quantity types. The function returns TRUE if this is the case and FALSE otherwise.

Express specification:

```
FUNCTION mgm_verify_transformation(
    an_item : mgm_any_meshed_geometric_item) : BOOLEAN;
  IF EXISTS(an_item.transformation) THEN
    IF 'MGM_ARM.MGM_AXIS_PLACEMENT' IN TYPEOF(an_item.transformation) THEN
      IF an_item.transformation.location.quantity_type :<>:
 mgm_get_context_quantity_type(an_item.containing_model, 'length') THEN
        RETURN(FALSE);
      END_IF;
    ELSE
      -- it is an mgm_axis_transformation_sequence
      REPEAT I := 1 TO SIZEOF(an_item.transformation.transformation_sequence);
        IF 'MGM_ARM.MGM_TRANSLATION' IN
 TYPEOF(an_item.transformation.transformation_sequence[i]) THEN
          IF an_item.transformation.transformation_sequence[i].quantity_type :<>:
 mgm_get_context_quantity_type(an_item.containing_model, 'length') THEN
            RETURN(FALSE);
          END_IF;
        ELSE
          -- it is an mgm_rotation
          IF an_item.transformation.transformation_sequence[i].quantity_type :<>:
              mgm_get_context_quantity_type(
              an_item.containing_model, 'plane_angle') THEN
            RETURN(FALSE);
          END_IF;
        END_IF;
      END_REPEAT;
    END_IF;
  END_IF;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- an_item specifies the mgm_any_meshed_geometric_item to be verified.

4.3.6.32 FUNCTION mgm_verify_acyclic_compound_meshed_geometric_item_tree

The function nrf_verify_acyclic_network_model_tree verifies that there is no circular reference in the tree of an mgm_compound_meshed_geometric_item instances. In other words, the tree forms an acyclic graph. The function returns TRUE if this is the case and FALSE otherwise.

Express specification:

```
FUNCTION mgm_verify_acyclic_compound_meshed_geometric_item_tree(
    an_item : mgm_any_meshed_geometric_item;
    super_items : LIST OF mgm_any_meshed_geometric_item) : BOOLEAN;

  IF 'MGM_ARM.MGM_COMPOUND_MESHED_GEOMETRIC_ITEM' IN TYPEOF(an_item) THEN
    REPEAT i := 1 TO SIZEOF(an_item.geometric_items);
      IF (an_item.geometric_items[i] IN super_items) THEN
        RETURN(FALSE);
      END_IF;
      IF NOT mgm_verify_acyclic_compound_meshed_geometric_item_tree(
          an_item.geometric_items[i], super_items + an_item.geometric_items) THEN
        RETURN(FALSE);
      END_IF;
    END_REPEAT;
  END_IF;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- an_item specifies the candidate mgm_any_meshed_geometric_item that is to be verified.
- super_items specifies the list of mgm_any_meshed_geometric_item instances that occur at the same or higher level as an_item.

4.3.6.33 FUNCTION mgm_verify_no_coincident_points

The function **mgm_verify_no_coincident_points** verifies that no two points in a given set of points coincide, that is the distance between each pair of points shall be greater than the given minimum_distance. The function returns TRUE if this is the case and FALSE otherwise.

Express specification:

```
FUNCTION mgm_verify_no_coincident_points(
    points : SET OF mgm_3d_cartesian_point;
    minimum_distance : REAL) : BOOLEAN;

  REPEAT i := 1 TO SIZEOF(points);
    REPEAT j := (i + 1) TO SIZEOF(points);
      IF mgm_compute_distance_between_points(points[i], points[j]) <= minimum_distance THEN
        RETURN(FALSE);
      END_IF;
    END_REPEAT;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- points specifies the set of mgm_3d_cartesian_point instances to verify, where each point shall use the same quantity_type for its x, y and z coordinates.
- mimimum_distance specifies the minimum distance that shall be present between each pair of points. It shall be specified in the same unit as the quantity_type of the coordinates of each of the points.

4.3.6.34 FUNCTION mgm_verify_no_colinear_points

The function **mgm_verify_no_colinear_points** verifies that the points in a given set are not co-linear. The function returns TRUE if this is the case and FALSE otherwise.

Express specification:

```
FUNCTION mgm_verify_no_colinear_points(
    points : SET OF mgm_3d_cartesian_point;
    tolerance : REAL) : BOOLEAN;
  LOCAL
    ux, uy, uz, ulen : REAL;
    vx, vy, vz, vlen : REAL;
    dot : REAL;
  END_LOCAL;
  IF SIZEOF(points) <= 2 THEN
    RETURN(TRUE);
  END_IF;
  REPEAT i := 3 TO SIZEOF(points);
    ux := points[i-1].x - points[i-2].x;
    uy := points[i-1].y - points[i-2].y;
    uz := points[i-1].z - points[i-2].z;
    ulen := SQRT(ux*ux + uy*uy + uz*uz);
    vx := points[i].x - points[i-1].x;
    vy := points[i].y - points[i-1].y;
    vz := points[i].z - points[i-1].z;
    vlen := SQRT(vx*vx + vy*vy + vz*vz);
    dot := (ux*vx + uy*vy + uz*vz) / (ulen * vlen);
    IF (dot <= (-1.0+tolerance)) OR (dot >= (1.0-tolerance)) THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- points specifies the set of mgm_3d_cartesian_point instances to verify, where each point shall use the same quantity_type for its x, y and z coordinates.
- tolerance specifies the applicable point coincidence uncertainty.

4.3.6.35 FUNCTION mgm_verify_quadrilateral

The function **mgm_verify_quadrilateral** verifies that an mgm_quadrilateral instance is valid by checking the following constraints: (1) The distance between each pair of points shall be greater than the applicable 'point_coincidence_length', (2) The quadrilateral shall be planar, (3) All four interior angles of the quadrilateral shall be greater than 0 and less than 180 degrees. The function returns TRUE if the quadrilateral is valid and FALSE if it is not.

Express specification:

```
FUNCTION mgm_verify_quadrilateral(
    quad : mgm_quadrilateral) : BOOLEAN;
  LOCAL
    tol              : REAL;
    dot              : REAL;
    mx, my, mz, mlen : REAL;
    nx, ny, nz, nlen : REAL;
    ux, uy, uz, ulen : REAL;
    vx, vy, vz, vlen : REAL;
  END_LOCAL;
  tol := mgm_get_context_uncertainty_value(
      quad.geometric_item.containing_model, 'point_coincidence_length');

  -- verify interior angle at p1 from vectors u and v
  -- u is vector p1 to p2, v is vector p1 to p4
  ux := quad.p2.x - quad.p1.x;
  uy := quad.p2.y - quad.p1.y;
  uz := quad.p2.z - quad.p1.z;
  vx := quad.p4.x - quad.p1.x;
  vy := quad.p4.y - quad.p1.y;
  vz := quad.p4.z - quad.p1.z;
  ulen := SQRT(ux*ux + uy*uy + uz*uz);
  vlen := SQRT(vx*vx + vy*vy + vz*vz);
  IF (ulen <= tol) OR (vlen <= tol) THEN
    RETURN(FALSE);
  END_IF;
  -- compute normalized dot product
  dot := (ux*vx + uy*vy + uz*vz) / (ulen * vlen);
  -- fail if angle to close to 0 or 180 deg
  IF (dot <= (-1.0+tol)) OR (dot >= (1.0-tol)) THEN
    RETURN(FALSE);
  END_IF;
  -- compute normal via cross-product
  nx := uy*vz - uz*vy;
  ny := uz*vx - ux*vz;
  nz := ux*vy - uy*vx;
  nlen := SQRT(nx*nx + ny*ny + nz*nz);
  IF nlen <= tol THEN
    RETURN(FALSE);
  END_IF;
  -- normalise n
  nx := nx / nlen;
  ny := ny / nlen;
  nz := nz / nlen;

  -- verify that p3 is in the plane spanned by p1, p2, p4
  -- m is vector from p1 to p3
  mx := quad.p3.x - quad.p1.x;
  my := quad.p3.y - quad.p1.y;
  mz := quad.p3.z - quad.p1.z;
  -- compute the length of the projection of m on n (i.e. absolute dot product)
  -- this may not be greater than the point_coincidence_length
  IF ABS(mx*nx + my*ny + mz*nz) > tol THEN
    RETURN(FALSE);
  END_IF;

  -- loop to verify correct interior angles at points p2, p3 ,p4
  REPEAT i := 2 TO 4;
    CASE i OF
    2: BEGIN
        -- verify interior angle at p2 from vectors u and v
        -- u is vector p2 to p3, v is vector p2 to p1
```



```
        ux := quad.p3.x - quad.p2.x;
        uy := quad.p3.y - quad.p2.y;
        uz := quad.p3.z - quad.p2.z;
        vx := quad.p1.x - quad.p2.x;
        vy := quad.p1.y - quad.p2.y;
        vz := quad.p1.z - quad.p2.z;
      END;
    3: BEGIN
        -- verify interior angle at p3 from vectors u and v
        -- u is vector p3 to p4, v is vector p3 to p2
        ux := quad.p4.x - quad.p3.x;
        uy := quad.p4.y - quad.p3.y;
        uz := quad.p4.z - quad.p3.z;
        vx := quad.p2.x - quad.p3.x;
        vy := quad.p2.y - quad.p3.y;
        vz := quad.p2.z - quad.p3.z;
      END;
    4: BEGIN
        -- verify interior angle at p4 from vectors u and v
        -- u is vector p4 to p1, v is vector p4 to p3
        ux := quad.p1.x - quad.p4.x;
        uy := quad.p1.y - quad.p4.y;
        uz := quad.p1.z - quad.p4.z;
        vx := quad.p3.x - quad.p4.x;
        vy := quad.p3.y - quad.p4.y;
        vz := quad.p3.z - quad.p4.z;
      END;
    END_CASE;
    ulen := SQRT(ux*ux + uy*uy + uz*uz);
    vlen := SQRT(vx*vx + vy*vy + vz*vz);
    IF (ulen <= tol) OR (vlen <= tol) THEN
      RETURN(FALSE);
    END_IF;
    -- compute normalized dot product
    dot := (ux*vx + uy*vy + uz*vz) / (ulen * vlen);
    -- fail if angle to close to 0 or 180 deg
    IF (dot <= (-1.0+tol)) OR (dot >= (1.0-tol)) THEN
      RETURN(FALSE);
    END_IF;
    -- compute normal via cross-product
    mx := uy*vz - uz*vy;
    my := uz*vx - ux*vz;
    mz := ux*vy - uy*vx;
    mlen := SQRT(mx*mx + my*my + mz*mz);
    IF mlen <= tol THEN
      RETURN(FALSE);
    END_IF;
    -- compute dot product for this normal and the reference normal at p1
    dot := mx*nx + my*ny + mz*nz;
    -- fail if dot negative because then one of the interior angles > 180 deg
    IF dot < 0.0 THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;

  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— quad specifies the candidate mgm_quadrilateral to be verified

4.3.6.36 FUNCTION mgm_verify_points_span_orthogonal_system

The function **mgm_verify_points_span_orthogonal_system** verifies that three cartesian points p1, p2 and p3 span a system of two orthogonal vectors p1 to p2 and p1 to p3. The function also verifies that points p2 and p3 do not coincide with point p1 within a given tolerance. The function returns TRUE if this is the case and FALSE otherwise.

Express specification:

```
FUNCTION mgm_verify_points_span_orthogonal_system(
    p1 : mgm_3d_cartesian_point;
    p2 : mgm_3d_cartesian_point;
    p3 : mgm_3d_cartesian_point;
    tolerance : REAL) : BOOLEAN;
  LOCAL
    v1x, v1y, v1z, v1len   : REAL;
    v2x, v2y, v2z, v2len   : REAL;
    normalized_dot_product : REAL;
  END_LOCAL;
  v1x := p2.x - p1.x;
  v1y := p2.y - p1.y;
  v1z := p2.z - p1.z;
  v2x := p3.x - p1.x;
  v2y := p3.y - p1.y;
  v2z := p3.z - p1.z;
  v1len := SQRT(v1x*v1x + v1y*v1y + v1z*v1z);
  v2len := SQRT(v2x*v2x + v2y*v2y + v2z*v2z);
  IF (v1len <= tolerance) OR (v2len <= tolerance) THEN
    RETURN(FALSE);
  END_IF;
  normalized_dot_product := (v1x*v2x + v1y*v2y + v1z*v2z)/(v1len*v2len);
  IF ABS(normalized_dot_product) > tolerance THEN
    RETURN(FALSE);
  END_IF;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- p1 specifies the first point of three that are to be verified to span an orthogonal system
- p2 specifies the second point of three that are to be verified to span an orthogonal system
- p3 specifies the third point of three that are to be verified to span an orthogonal system
- tolerance specifies the minimum length tolerance to take into account

4.3.6.37 FUNCTION mgm_verify_points_use_context_length_quantity_type

The function **mgm_verify_points_use_context_length_quantity_type** verifies that the coordinates of one or more mgm_3d_cartesian_point instances all use the 'length' quantity type that is applicable for a given mgm_meshed_geometric_model. The function returns TRUE if this is the case, and FALSE if not.

Express specification:

```
FUNCTION mgm_verify_points_use_context_length_quantity_type(
    points : SET OF mgm_3d_cartesian_point;
    a_model : mgm_meshed_geometric_model) : BOOLEAN;
LOCAL
  context_length_quantity_type : nrf_any_quantity_type;
```



```
END_LOCAL;
  context_length_quantity_type := mgm_get_context_quantity_type(
      a_model, 'length');
  REPEAT i := 1 TO SIZEOF(points);
    IF points[i].quantity_type :<>: context_length_quantity_type THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- points specifies the set of mgm_3d_cartesian_point instances to be verified
- a_model specifies the mgm_meshed_geometric_model from which to get the applicable 'length' quantity type for the coordinates of all points.

4.3.6.38 FUNCTION mgm_verify_surface_grid_spacings

The function **mgm_verify_surface_grid_spacings** verifies that the grid spacings for an mgm_meshed_primitive_bounded_surface are valid. It is required that the first value is zero, the last value is one and all values are in ascending order. The function returns TRUE if this is the case and FALSE otherwise.

Express specification:

```
FUNCTION mgm_verify_surface_grid_spacings(
    a_grid_spacings : LIST OF REAL) : BOOLEAN;
  IF a_grid_spacings[LOINDEX(a_grid_spacings)] <> 0.0 THEN
    RETURN(FALSE);
  END_IF;
  IF a_grid_spacings[HIINDEX(a_grid_spacings)] <> 1.0 THEN
    RETURN(FALSE);
  END_IF;
  REPEAT i := 1 TO SIZEOF(a_grid_spacings) - 1;
    IF (a_grid_spacings[i] >= a_grid_spacings[i+1]) THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

- a_grid_spacings specifies the list of grid spacing values to be verified.

4.3.6.39 FUNCTION mgm_verify_start_and_end_angles

The function **mgm_verify_start_and_end_angles** verifies the following constraints on the start_angle and end_angle for surfaces or solids of revolution.

- The end_angle is required to be greater than the start_angle.
- Both angles are required to lie in the interval from -360 to +360 degrees.
- The difference between end_angle and start_angle is required to be 360 degrees or less.

For all these constraints a numerical tolerance is taken into account that is related to the applicable point_coincidence_length uncertainty. The function returns TRUE if the constraints are met and FALSE otherwise.

Express specification:

```
FUNCTION mgm_verify_start_and_end_angles(
    start_angle : REAL;
    end_angle   : REAL;
    tolerance   : REAL) : BOOLEAN;
  LOCAL
    tol : REAL;
  END_LOCAL;
  -- convert point_coincidence_length uncertainty to angular tolerance in degree
  tol := tolerance * 180.0 / PI;
  IF {(-tol - 360.0) <= start_angle < (360.0 - tol)} AND
     {(-360.0 + tol) < end_angle <= (360.0 + tol)} AND
     ((start_angle + tol) < end_angle) AND
     ((end_angle - start_angle) <= (360.0 + tol)) THEN
    RETURN(TRUE);
  ELSE
    RETURN(FALSE);
  END_IF;
END_FUNCTION;
```

Argument definitions:

- start_angle specifies the start angle (in degree) of the surface or solid of revolution to be verified
- end_angle specifies the end angle (in degree) of the surface or solid of revolution to be verified
- tolerance specifies the applicable 'point_coincidence_uncertainty'

4.3.6.40 FUNCTION mgm_verify_solid_box

The function **mgm_verify_solid_box** verifies whether the points of an mgm_solid_box define a valid solid box according to the following criteria: (1) the points p1, p2, p3, p4 do not coincide, (2) the points p1, p2, p3 span an orthogonal system, (3) vector p1 to p4 has the same direction as the positive normal on the plane spanned by vector p1 to p2 and vector p1 to p3.

The function returns TRUE if this is the case and FALSE otherwise.

Express specification:

```
FUNCTION mgm_verify_solid_box(
    sb : mgm_solid_box) : BOOLEAN;
  LOCAL
    tol : REAL;
    mx, my, mz, mlen : REAL;
    nx, ny, nz, nlen : REAL;
    ux, uy, uz : REAL;
    vx, vy, vz : REAL;
  END_LOCAL;
  tol := mgm_get_context_uncertainty_value(
    sb.geometric_item.containing_model, 'point_coincidence_length');

  IF NOT mgm_verify_no_coincident_points([sb.p1, sb.p2, sb.p3, sb.p4], tol) THEN
    RETURN(FALSE);
  END_IF;
```



```
  IF NOT mgm_verify_points_span_orthogonal_system(sb.p1, sb.p2, sb.p3, tol) THEN
    RETURN(FALSE);
  END_IF;

  -- u is vector p1 to p2, v is vector p1 to p3
  ux := sb.p2.x - sb.p1.x;
  uy := sb.p2.y - sb.p1.y;
  uz := sb.p2.z - sb.p1.z;
  vx := sb.p3.x - sb.p1.x;
  vy := sb.p3.y - sb.p1.y;
  vz := sb.p3.z - sb.p1.z;

  -- compute normal via cross product of u and v
  nx := uy*vz - uz*vy;
  ny := uz*vx - ux*vz;
  nz := ux*vy - uy*vx;
  nlen := SQRT(nx*nx + ny*ny + nz*nz);
  IF nlen <= tol THEN
    RETURN(FALSE);
  END_IF;

  -- m is vector from p1 to p4
  mx := sb.p4.x - sb.p1.x;
  my := sb.p4.y - sb.p1.y;
  mz := sb.p4.z - sb.p1.z;
  mlen := SQRT(mx*mx + my*my + mz*mz);
  IF mlen <= tol THEN
    RETURN(FALSE);
  END_IF;

  -- vector p1 to p4 must be parallel to the normal n
  -- i.e. the normalised dot product must be greater than 1-tol
  IF NOT ((mx*nx + my*ny + mz*nz)/(mlen*nlen) > (1.0 - tol)) THEN
    RETURN(FALSE);
  END_IF;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— sb specifies the mgm_solid_box to be verified.

4.3.6.41 FUNCTION mgm_verify_solid_triangular_prism

The function **mgm_verify_solid_triangular_prism** verifies whether the points of an mgm_solid_triangular_prism define a valid solid triangular prism according to the following criteria: (1) the points p1, p2, p3 do not coincide, (2) the points p1, p2, p3 are not colinear, (3) vector p1 to p4 has the same direction as the positive normal on the plane spanned by vector p1 to p2 and vector p1 to p3.

The function returns TRUE if this is the case and FALSE otherwise.

Express specification:

```
FUNCTION mgm_verify_solid_triangular_prism(
    stp : mgm_solid_triangular_prism) : BOOLEAN;
  LOCAL
    tol : REAL;
    mx, my, mz, mlen : REAL;
    nx, ny, nz, nlen : REAL;
    ux, uy, uz : REAL;
```



```
    vx, vy, vz : REAL;
  END_LOCAL;
  tol := mgm_get_context_uncertainty_value(
    stp.geometric_item.containing_model, 'point_coincidence_length');

  IF NOT mgm_verify_no_coincident_points([stp.p1, stp.p2, stp.p3, stp.p4], tol) THEN
    RETURN(FALSE);
  END_IF;

  IF NOT mgm_verify_no_colinear_points([stp.p1, stp.p2, stp.p3], tol) THEN
    RETURN(FALSE);
  END_IF;

  -- u is vector p1 to p2, v is vector p1 to p3
  ux := stp.p2.x - stp.p1.x;
  uy := stp.p2.y - stp.p1.y;
  uz := stp.p2.z - stp.p1.z;
  vx := stp.p3.x - stp.p1.x;
  vy := stp.p3.y - stp.p1.y;
  vz := stp.p3.z - stp.p1.z;
  -- compute normal via cross product of u and v
  nx := uy*vz - uz*vy;
  ny := uz*vx - ux*vz;
  nz := ux*vy - uy*vx;
  nlen := SQRT(nx*nx + ny*ny + nz*nz);
  IF nlen <= tol THEN
    RETURN(FALSE);
  END_IF;
  -- m is vector from p1 to p4
  mx := stp.p4.x - stp.p1.x;
  my := stp.p4.y - stp.p1.y;
  mz := stp.p4.z - stp.p1.z;
  mlen := SQRT(mx*mx + my*my + mz*mz);
  IF mlen <= tol THEN
    RETURN(FALSE);
  END_IF;
  -- vector p1 to p4 must be parallel to the normal n
  -- i.e. the normalised dot product must be greater than 1-tol
  IF NOT ((mx*nx + my*ny + mz*nz)/(mlen*nlen) > (1.0 - tol)) THEN
    RETURN(FALSE);
  END_IF;

  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— stp specifies the mgm_solid_triangular_prism to be verified.

4.3.6.42 RULE mgm_verify_referencing_of_faces

The RULE **mgm_verify_referencing_of_faces** verifies that any mgm_face that is part of any mgm_meshed_geometric_model is referenced only once.

Express specification:

```
RULE mgm_verify_referencing_of_faces FOR (nrf_root);
LOCAL
  rule_satisfied : LOGICAL := TRUE;
  all_faces : LIST OF mgm_face := [];
  root_model : nrf_network_model;
```



```
  check_face : mgm_face;
END_LOCAL;
  REPEAT i := 1 TO SIZEOF(nrf_root[1].root_models);
    root_model := nrf_root[1].root_models[i];
    IF 'MGM_ARM.MGM_MESHED_GEOMETRIC_MODEL' IN TYPEOF(root_model) THEN
      all_faces := all_faces +
 mgm_get_faces_from_meshed_geometric_item(root_model.root_item);
    END_IF;
  END_REPEAT;
  REPEAT WHILE ((SIZEOF(all_faces) > 1) AND rule_satisfied);
    check_face := all_faces[1];
    REMOVE(all_faces, 1);
    IF check_face IN all_faces THEN
      rule_satisfied := FALSE;
    END_IF;
  END_REPEAT;
WHERE
  wr1: rule_satisfied;
END_RULE;
```

4.3.6.43 FUNCTION mgm_get_faces_from_meshed_geometric_item

The function **mgm_get_faces_from_meshed_geometric_item** returns a list of all mgm_face instances that are part of a given mgm_any_meshed_geometric_item.

Express specification:

```
FUNCTION mgm_get_faces_from_meshed_geometric_item (
    an_item : mgm_any_meshed_geometric_item) : LIST OF mgm_face;
  LOCAL
    all_faces: LIST OF mgm_face := [];
  END_LOCAL;
  IF 'MGM_ARM.MGM_MESHED_PRIMITIVE_BOUNDED_SURFACE' IN TYPEOF(an_item) THEN
    IF EXISTS(an_item.side1_faces) THEN
      all_faces := all_faces + an_item.side1_faces;
    END_IF;
    IF EXISTS(an_item.side2_faces) THEN
      all_faces := all_faces + an_item.side2_faces;
    END_IF;
  ELSE
    IF 'MGM_ARM.MGM_COMPOUND_MESHED_GEOMETRIC_ITEM' IN TYPEOF(an_item) THEN
      REPEAT i := 1 TO SIZEOF(an_item.geometric_items);
        all_faces := all_faces +
 mgm_get_faces_from_meshed_geometric_item(an_item.geometric_items[i]);
      END_REPEAT;
    ELSE
      IF 'MGM_ARM.MGM_MESHED_BOOLEAN_DIFFERENCE_SURFACE' IN TYPEOF(an_item) THEN
        all_faces := all_faces +
 mgm_get_faces_from_meshed_geometric_item(an_item.base_surface);
      ELSE
        IF 'MGM_ARM.MGM_MESHED_GEOMETRIC_ITEM_BY_SUBMODEL' IN TYPEOF(an_item) THEN
          all_faces := all_faces +
 mgm_get_faces_from_meshed_geometric_item(an_item.submodel.root_item);
        END_IF;
      END_IF;
    END_IF;
  END_IF;
  RETURN(all_faces);
END_FUNCTION;
```

Argument definitions:

— an_item specifies the mgm_any_meshed_geometric_item for which all contained mgm_face instances should be returned.

4.3.6.44 FUNCTION mgm_verify_enclosure_faces

The function **mgm_verify_enclosure_faces** verifies that all surfaces of an mgm_enclosure are part of the containing_model, either directly or through a submodel and that the active side as specified in active_sides of an mgm_enclosure shall be one or both of the actives side specified for the corresponding surface. If this is the case the function returns TRUE, else FALSE.

Express specification:

```
FUNCTION mgm_verify_enclosure_faces(
    an_enclosure : mgm_enclosure) : BOOLEAN;
  -- !!TBD!!
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— an_enclosure specifies the mgm_enclosure to be verified.

4.3.7 MGM meshed boolean construction geometry UoF

The MGM meshed boolean construction geometry UoF contains application objects that enable the representation of meshed geometric items resulting from boolean construction operations.

4.3.7.1 TYPE mgm_half_space_selector_type

The **mgm_half_space_selector_type** specifies which half space as defined by an mgm_primitive_solid to use in boolean construction operations. When the **mgm_half_space_selector_type** is INSIDE, the half space inside the mgm_primitive_solid or tas_primitive_half_space is used as a boolean operand. Likewise a value of OUTSIDE indicates the half space outside the mgm_primitive_solid or tas_primitive_half_space is used as a boolean operand.

Express specification:

```
TYPE mgm_half_space_selector_type = ENUMERATION OF (
  INSIDE,
  OUTSIDE);
END_TYPE;
```

4.3.7.2 ENTITY tas_half_space_solid

An mgm_half_space_solid specifies a half space solid shape for the purpose of boolean construction operations. A half space solid is one of the two continuous subspaces that result from dividing the full 3D space into exactly two parts.

Express specification:

```
ENTITY mgm_half_space_solid;
  id                    : nrf_identifier;
  name                  : nrf_label;
```



```
  description          : nrf_text;
  transformation       : OPTIONAL mgm_axis_transformation;
  solid                : mgm_primitive_solid;
  half_space_selector : mgm_half_space_selector_type;
END_ENTITY;
```

Attribute definitions:

- id specifies the identifier of an mgm_half_space_solid.
- name specifies the human-interpretable name of an mgm_half_space_solid.
- description specifies the textual description of an mgm_half_space_solid.
- transformation optionally specifies the mgm_axis_transformation to be applied to the solid. The transformation is defined with respect to the placement of the mgm_meshed_geometric_item that uses this mgm_half_space_solid.
- solid specifies the mgm_primitive_solid of which the volume is used to define the half space.
- half_space_selector specifies whether the inside or the outside half of the solid constitutes the substance of the solid. The definitions of what constitutes INSIDE and OUTSIDE are explicitly given for each of the subtypes of mgm_primitive_solid. A half space is one of the two continuous subspaces that result from dividing the full 3D space into exactly two parts.

4.3.7.3 ENTITY mgm_meshed_boolean_difference_surface

An **mgm_meshed_boolean_difference_surface** specifies a bounded surface that is defined by a base_surface of which the part enclosed by a given cutting_solid is removed. In other words the resulting surface after the boolean operation contains all points of the base_surface that are outside the half space defined by the cutting_solid.

Express specification:

```
ENTITY mgm_meshed_boolean_difference_surface
  SUBTYPE OF(mgm_any_meshed_geometric_item);
  base_surface                 : mgm_any_meshed_geometric_item;
  cutting_solid                : mgm_half_space_solid;
WHERE
  wr1: mgm_verify_boolean_difference_base_surface(base_surface);
  wr2: cutting_solid.solid.geometric_item :=: SELF;
END_ENTITY;
```

Attribute definitions:

- base_surface specifies the bounded surface that is the first operand of the boolean difference operation.
- cutting_solid specifies the mgm_primitive_solid that is the second, cutting half space, operand of the boolean difference operation.

Formal propositions:

- wr1: The leaf items of the base_surface shall all be mgm_meshed_primitive_bounded_surface or **mgm_meshed_boolean_difference_surface** instances.
- wr2: The geometric_item of the solid of the cutting_solid shall reference this **mgm_meshed_boolean_difference_surface**.

4.3.7.4 FUNCTION mgm_verify_boolean_difference_base_surface

The function **mgm_verify_boolean_difference_base_surface** verifies that the leaf items of the base_surface of a given mgm_meshed_boolean_difference_surface are all mgm_meshed_primitive_bounded_surface or mgm_meshed_boolean_difference_surface instances. The function returns TRUE if this is the case and FALSE otherwise.

Express specification:

```
FUNCTION mgm_verify_boolean_difference_base_surface(
    a_base_surface : mgm_any_meshed_geometric_item) : BOOLEAN;
  IF 'MGM_ARM.MGM_COMPOUND_MESHED_GEOMETRIC_ITEM' IN TYPEOF(a_base_surface) THEN
    REPEAT i := 1 TO SIZEOF(a_base_surface.geometric_items);
      IF NOT mgm_verify_boolean_difference_base_surface(
          a_base_surface.geometric_items[i]) THEN
        RETURN(FALSE);
      END_IF;
    END_REPEAT;
  ELSE
    IF NOT (
 ('MGM_ARM.MGM_MESHED_PRIMITIVE_BOUNDED_SURFACE' IN TYPEOF(a_base_surface)) OR
 ('MGM_ARM.MGM_MESHED_BOOLEAN_DIFFERENCE_SURFACE' IN TYPEOF(a_base_surface)))
    THEN
      RETURN(FALSE);
    END_IF;
  END_IF;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— a_base_surface specifies the mgm_any_meshed_geometric_item to be verified.

4.3.8 END_SCHEMA declaration for mgm_arm

The following EXPRESS declaration ends the mgm_arm schema.

Express specification:

```
END_SCHEMA; -- mgm_arm
```

4.4 Space kinematic model (SKM) module

This subclause specifies the unit of functionality for the space kinematic model module. There is only one:

1. SKM rigid body kinematics UoF

4.4.1 SCHEMA declaration for skm_arm

The following EXPRESS declaration begins the skm_arm schema.

Express specification:


```
SCHEMA skm_arm;
-- $Id$
-- Copyright (c) 1995-2018 European Space Agency (ESA)
-- All rights reserved.
```

4.4.2 Interfaced schema(ta) for skm_arm

The mgm_arm schema uses the nrf_arm schema specified in [STEP-NRF] and the mgm_arm schema.

Express specification:

```
USE FROM nrf_arm;
USE FROM mgm_arm;
```

4.4.3 CONSTANT specifications

Two constants are defined for the SKM module:

Express specification:

```
CONSTANT
  SCHEMA_OBJECT_IDENTIFIER : STRING :=
 '{http://www.purl.org/ESA/step-tas/v6.0/skm_arm.exp}';
  -- in formal version to be replaced with
  -- '{ iso standard n part(p) version(v) }'
END_CONSTANT;
```

Constant definitions:

SCHEMA_OBJECT_IDENTIFIER provides a built-in way to reference the object identifier of the protocol for version verification. For the definition and usage of the object identifier see ISO 10303-1 and Annex E.

4.4.4 SKM rigid body kinematics UoF

The SKM rigid body kinematics UoF provides the objects needed to add rigid body kinematics definitions to a meshed geometric model specified by an mgm_meshed_geometric_model. The basic object is the skm_kinematic_joint which defines the potential movement of one part of a geometric model (a complete subtree) with respect to its containing part, which is the next higher level mgm_compound_meshed_geometric_item.

4.4.4.1 ENTITY skm_kinematic_degree_of_freedom

An **skm_kinematic_degree_of_freedom** is an abstract supertype that provides a generic mechanism to specify a sliding (i.e. translational) or revolute (i.e. rotational) degree of freedom for a kinematic joint.

Express specification:

```
ENTITY skm_kinematic_degree_of_freedom
  ABSTRACT SUPERTYPE OF( ONEOF(
    skm_sliding_degree_of_freedom,
    skm_revolute_degree_of_freedom));
  axis : mgm_3d_direction;
END_ENTITY;
```

Attribute definitions:

— axis specifies the translation direction or the axis of rotation for the kinematic degree of freedom.

4.4.4.2 ENTITY skm_sliding_degree_of_freedom

An **skm_sliding_degree_of_freedom** is a kind of skm_kinematic_degree_of_freedom that specifies a sliding degree of freedom for a kinematic joint and optionally the stops that constrain the allowable range for the translation distance. The positive sliding distance is defined in the direction of the translation axis specified in the axis attribute that is inherited from skm_kinematic_degree_of_freedom. Zero sliding distance is defined as the position of the affected geometric item after any static transformations have been applied.

Express specification:

```
ENTITY skm_sliding_degree_of_freedom
  SUBTYPE OF(skm_kinematic_degree_of_freedom);
  lower_stop_distance : OPTIONAL nrf_real_quantity_value_prescription;
  upper_stop_distance : OPTIONAL nrf_real_quantity_value_prescription;
WHERE
  wr1: (NOT EXISTS(lower_stop_distance)) OR (
    nrf_verify_dimensional_exponents(
 lower_stop_distance.quantity_type.quantity_category,
      1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0));
  wr2: (NOT EXISTS(upper_stop_distance )) OR (
    nrf_verify_dimensional_exponents(
 upper_stop_distance.quantity_type.quantity_category,
      1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0));
END_ENTITY;
```

Attribute definitions:

— lower_stop_distance optionally specifies the allowed lower limit sliding distance for a sliding kinematic joint.

— upper_stop_distance optionally specifies the allowed upper limit sliding distance for a sliding kinematic joint.

Formal propositions:

— wr1: If lower_stop_distance exists it shall have the dimension of length.

— wr2: If upper_stop_distance exists it shall have the dimension of length.

4.4.4.3 ENTITY skm_revolute_degree_of_freedom

An **skm_revolute_degree_of_freedom** is a kind of skm_kinematic_degree_of_freedom that specifies a revolute degree of freedom for a kinematic joint and optionally the stops (or limit angles) that constrain the allowable range for the rotation angle. The positive rotation angle is defined with respect to the rotation axis specified in the axis attribute that is inherited from skm_kinematic_degree_of_freedom. Zero rotation is defined as the orientation of the affected geometric item after any static transformations have been applied.

Express specification:

```
ENTITY skm_revolute_degree_of_freedom
  SUBTYPE OF(skm_kinematic_degree_of_freedom);
  lower_stop_angle        : OPTIONAL nrf_real_quantity_value_prescription;
```



```
  upper_stop_angle        : OPTIONAL nrf_real_quantity_value_prescription;
WHERE
  wr1: (NOT EXISTS(lower_stop_angle)) OR
 (lower_stop_angle.quantity_type.quantity_category.name = 'plane_angle');
  wr2: (NOT EXISTS(upper_stop_angle)) OR
 (upper_stop_angle.quantity_type.quantity_category.name = 'plane_angle');
END_ENTITY;
```

Attribute definitions:

- lower_stop_angle optionally specifies the allowed lower limit angle of rotation for a revolute kinematic joint.
- upper_stop_angle optionally specifies the allowed upper limit angle of rotation for a revolute kinematic joint.

Formal propositions:

- wr1: If lower_stop_angle exists it shall have the quantity type 'plane_angle' and
- wr2: If upper_stop_angle exists it shall have the quantity type 'plane_angle' and its value shall be in the range from –360 to +360 degrees.

4.4.4.4 ENTITY skm_kinematic_joint

An **skm_kinematic_joint** specifies the possibility for the movement of a rigid body represented by an mgm_any_meshed_geometric_item with respect to its containing (next higher level) geometric item or model. The kinematic joint may have up to six degrees of freedom, comprising at most three sliding degrees of freedom and at most three revolute degrees of freedom.

NOTE In engineering analysis applications support for rigid body kinematics is known under variety of names, among others: kinematic joints, moving, articulated or animated bodies, shapes or assemblies.

Express specification:

```
ENTITY skm_kinematic_joint;
  geometric_item     : mgm_any_meshed_geometric_item;
  degrees_of_freedom : LIST [1:6] OF skm_kinematic_degree_of_freedom;
WHERE
  wr1: skm_verify_degrees_of_freedom(SELF);
END_ENTITY;
```

Attribute definitions:

- geometric_item specifies the mgm_any_meshed_geometric_item for which the kinematic joint is defined.
- degrees_of_freedom specifies the sequence of up to six kinematic degrees of freedom for the joint.

Formal propositions:

- wr1: Verify for the following constraints: (1) The number of degrees shall not be more than six, (2) For each degree of freedom of type skm_sliding_degree_of_freedom for which both lower_stop_distance and upper_stop_distance are specified, the value of lower_stop_distance shall be less than the value of upper_stop_distance. (3) For each degree of freedom of type skm_revolute_degree_of_freedom for which lower_stop_angle is specified, the value of

lower_stop_angle shall range from –360 to +360 degrees.(4) For each degree of freedom of type skm_revolute_degree_of_freedom for which upper_stop_angle is specified, the value of upper_stop_angle shall range from –360 to +360 degrees.(5) For each degree of freedom of type skm_revolute_degree_of_freedom for which both lower_stop_angle and upper_stop_angle are specified, the value of lower_stop_angle should be less than the value of upper_stop_angle.(6) The number of degrees of freedom of type skm_sliding_degree_of_freedom shall not be more than three. (7)The number of degrees of freedom of type skm_revolute_degree_of_freedom shall not be more than three.

NOTE 1: Degrees of freedom not listed in degrees_of_freedom are considered to be fixed.

NOTE 2: The coordinate transformations resulting from translations and rotations along the axes as defined by the degrees of freedom have to be applied in the order in which these are defined in degrees_of_freedom. These transformations have to be applied in the same was as is documented under mgm_axis_transformation_sequence. In this context the rotations originating from instances of skm_revolute_degree_of_freedom have to be treated in the same way as the rotations specified by instances of mgm_rotation_with_axes_moving.

4.4.4.5 FUNCTION skm_verify_degrees_of_freedom

The function **skm_verify_degrees_of_freedom** verifies that (1) The number of degrees shall not be more than six, (2) For each degree of freedom of type skm_sliding_degree_of_freedom for which both lower_stop_distance and upper_stop_distance are specified, the value of lower_stop_distance shall be less than the value of upper_stop_distance. (3) For each degree of freedom of type skm_revolute_degree_of_freedom for which lower_stop_angle is specified, the value of lower_stop_angle shall range from –360 to +360 degrees.(4) For each degree of freedom of type skm_revolute_degree_of_freedom for which upper_stop_angle is specified, the value of upper_stop_angle shall range from –360 to +360 degrees.(5) For each degree of freedom of type skm_revolute_degree_of_freedom for which both lower_stop_angle and upper_stop_angle are specified, the value of lower_stop_angle should be less than the value of upper_stop_angle.(6) The number of degrees of freedom of type skm_sliding_degree_of_freedom shall not be more than three. (7)The number of degrees of freedom of type skm_revolute_degree_of_freedom shall not be more than three.

Express specification:

```
FUNCTION skm_verify_degrees_of_freedom(
           a_joint : skm_kinematic_joint) : BOOLEAN;
  LOCAL
    dist_tol  : REAL;
    angle_tol : REAL;
    dof_count : INTEGER;
    a_dof     : skm_kinematic_degree_of_freedom;
    sliders   : LIST OF skm_sliding_degree_of_freedom;
    rotators  : LIST OF skm_revolute_degree_of_freedom;
  END_LOCAL;
  dist_tol := mgm_get_context_uncertainty_value(
      a_joint.geometric_item.containing_model, 'point_coincidence_length');
  angle_tol := dist_tol * 180.0 / PI;
  sliders := [];
  rotators := [];
  REPEAT dof_count := 1 TO SIZEOF(a_joint.degrees_of_freedom);
    a_dof := a_joint.degrees_of_freedom[dof_count];
    IF 'SKM_ARM.SKM_SLIDING_DEGREE_OF_FREEDOM' IN TYPEOF(a_dof) THEN
       sliders := sliders + [a_dof];
       IF (EXISTS(a_dof.lower_stop_distance)) AND
          (EXISTS(a_dof.upper_stop_distance)) THEN
```



```
        IF ((a_dof.upper_stop_distance.val -
             a_dof.lower_stop_distance.val) < dist_tol) THEN
          RETURN(FALSE);
        END_IF;
      END_IF;
    ELSE
      -- it is a skm_revolute_degree_of_freedom
      rotators := rotators + [a_dof];
      IF EXISTS(a_dof.lower_stop_angle) THEN
        IF (a_dof.lower_stop_angle.val < (-360.0 - angle_tol)) OR
           ((360.0 + angle_tol) < a_dof.lower_stop_angle.val) THEN
          RETURN(FALSE);
        END_IF;
      END_IF;
      IF EXISTS(a_dof.upper_stop_angle) THEN
        IF (a_dof.upper_stop_angle.val < (-360.0 - angle_tol)) OR
           ((360.0 + angle_tol) < a_dof.upper_stop_angle.val) THEN
          RETURN(FALSE);
        END_IF;
      END_IF;
      IF (EXISTS(a_dof.lower_stop_angle)) AND
         (EXISTS(a_dof.upper_stop_angle)) THEN
        IF ((a_dof.upper_stop_angle.val -
             a_dof.lower_stop_angle.val) < angle_tol) THEN
          RETURN(FALSE);
        END_IF;
      END_IF;
    END_IF;
  END_REPEAT;
  IF SIZEOF(sliders) > 3 THEN
    RETURN(FALSE);
  END_IF;
  IF SIZEOF(rotators) > 3 THEN
    RETURN(FALSE);
  END_IF;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— a_joint specifies the skm_kinematic_joint of which the degrees of freedom are to be verified.

4.4.5 END_SCHEMA declaration for skm_arm

The following EXPRESS declaration ends the skm_arm schema.

Express specification:

```
END_SCHEMA; -- skm_arm
```

4.5 Space mission aspects (SMA) module

This subclause specifies the units of functionality for the space mission aspects module. There is only one:

1. SMA space mission aspects UoF

4.5.1 SCHEMA declaration for sma_arm

The following EXPRESS declaration begins the sma_arm schema.

Express specification:

```
SCHEMA sma_arm;
-- $Id$
-- Copyright (c) 1995-2018 European Space Agency (ESA)
-- All rights reserved.
```

4.5.2 Interfaced schema(ta) for sma_arm

The mgm_arm schema uses the nrf_arm schema specified in [STEP-NRF] and the mgm_arm and skm_arm schemata.

Express specification:

```
USE FROM nrf_arm;
USE FROM mgm_arm;
USE FROM skm_arm;
```

4.5.3 CONSTANT specifications

One constant is defined for the SMA module: SCHEMA_OBJECT_IDENTIFIER.

Express specification:

```
CONSTANT
  SCHEMA_OBJECT_IDENTIFIER : STRING :=
 '{http://www.purl.org/ESA/step-tas/v6.0/sma_arm.exp}';
  -- in formal version to be replaced with
  -- '{ iso standard n part(p) version(v) }'
END_CONSTANT;
```

Constant definitions:

SCHEMA_OBJECT_IDENTIFIER provides a built-in way to reference the object identifier of the protocol for version verification. For the definition and usage of the object identifier see ISO 10303-1 and Annex E.

4.5.4 SMA space mission aspects UoF

The SMA space mission aspects UoF contains the application objects to define all aspects related to the mission and the environment for the object to be analysed, simulated, tested or operated. These aspects include space coordinate system and directions, orbit, orientation, pointing, events and environmental parameters of celestial bodies and interplanetary space.

4.5.4.1 ENTITY sma_space_mission_case

An **sma_space_mission_case** is a type of nrf_case that defines an analysis, simulation, test or operation case for a space mission. It may specify subcases to represent a sequence of mission phases. An **sma_space_mission_case** defines the applicable space coordinate system, optionally an epoch that sets the

reference for zero mission elapsed time, kinematic articulations if applicable, optionally an orbit_arc and optionally a set of parameters that characterize the applicable space environment.

An **sma_space_mission_case** may define a list of subcases that would be executed in sequence to represent a chain of mission phases. Subcases inherit the settings from their parent case but may override their settings with subcase specific ones.

The location and orientation of the main body of the object to be analyzed, simulated, tested or operated (typically a spacecraft), is specified by an sma_kinematic_articulation for the root_item of an mgm_meshed_geometric_model.

Express specification:

```
ENTITY sma_space_mission_case
  SUBTYPE OF(nrf_case);
  SELF\nrf_case.subcases : LIST OF UNIQUE sma_space_mission_case;
  coordinate_system      : sma_space_coordinate_system;
  reference_epoch        : OPTIONAL nrf_date_and_time;
  articulations          : LIST OF UNIQUE sma_kinematic_articulation;
  orbit_arc              : OPTIONAL sma_orbit_arc;
  sun_direction          : OPTIONAL sma_pointing_to_star;
  space_environment      : OPTIONAL sma_space_environment;
WHERE
  wr1: sma_verify_kinematic_articulations(SELF);
END_ENTITY;
```

Attribute definitions:

- subcases specifies a list of next lower level **sma_space_mission_case** instances.
- coordinate_system specifies the reference space coordinate system for an **sma_space_mission_case**.
- reference_epoch optionally specifies the date and time which is the reference instance in time for zero mission elapsed time.
- articulations specifies a list of sma_kinematic_articulation instances that define dynamic coordinate system transformations for kinematic joints, i.e. rigid body kinematic movements to be applied to a geometric item with respect to the coordinate reference system of its next higher level geometric item or model. Articulations are applied in addition to any static coordinate system transformation specified in the transformation attribute of the mgm_any_meshed_geometric_item for which a kinematic joint is defined.
- orbit_arc optionally specifies a (part of a) trajectory in space.
- space_environment optionally specifies a collection of parameters that characterize the space environment external to the object that is represented by the nrf_network_model that is referenced in the for_model attribute.

Formal propositions:

- wr1: The articulations shall use appropriate quantity types.

4.5.4.2 ENTITY sma_space_coordinate_system

An **sma_space_coordinate_system** specifies a reference coordinate system with respect to one or more relevant celestial bodies for use in the definition of the trajectory and the orientation of the meshed geometric model that represents a spacecraft (or another object in space).

Express specification:

```
ENTITY sma_space_coordinate_system;
  reference_system   : nrf_enumeration_quantity_value_literal;
  inertial_direction : OPTIONAL nrf_enumeration_quantity_value_literal;
WHERE
  wr1: reference_system.quantity_type.name = 'space_coordinate_system_type';
  wr2: (NOT EXISTS(inertial_direction)) OR
    (inertial_direction.quantity_type.name = 'standard_direction_in_space');
END_ENTITY;
```

Attribute definitions:

- reference_system specifies an applicable space coordinate system type.
- inertial_direction optionally specifies an applicable standard direction in space.

Formal propositions:

- wr1: The quantity_type of the reference_system shall be a 'space_coordinate_system_type'.
- wr2: If it exists, the quantity_type of the inertial_direction shall be a 'standard_direction_in_space'.

Informal propositions:

ip1: The inertial_direction is only relevant for a reference_system that is (or is considered to be) inertial in space.

4.5.4.3 ENTITY sma_orbit_arc

An **sma_orbit_arc** is an abstract supertype that specifies the generic part of the data necessary for the specification of an orbit arc. It provides a generic mechanism to reference an sma_discretized_orbit_arc or an sma_keplerian_orbit_arc.

Express specification:

```
ENTITY sma_orbit_arc
  ABSTRACT SUPERTYPE OF(ONEOF(sma_discretized_orbit_arc, sma_keplerian_orbit_arc));
  orbit_class                 : nrf_enumeration_quantity_value_literal;
  governing_celestial_body    : OPTIONAL sma_celestial_body;
  orbit_period                : OPTIONAL nrf_real_quantity_value_prescription;
WHERE
  wr1: orbit_class.quantity_type.name = 'orbit_class';
  wr2: nrf_verify_dimensional_exponents(
    orbit_period.quantity_type.quantity_category,
    0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0);
END_ENTITY;
```

Attribute definitions:

- orbit_class specifies an orbit class name, such as 'general', 'geostationary', 'sun_synchronous', 'molniya'.
- governing_celestial_body optionally specifies the celestial body that governs the orbit arc.
- orbit_period optionally specifies the period of the orbit arc in case it is a closed arc, probably a circular or an elliptical arc.

Formal propositions:

- wr1: The quantity_type of the orbit_class shall be named 'orbit_class'.
- wr2: The dimension of orbit_period shall be 'time'.

4.5.4.4 ENTITY sma_orbit_position_and_velocity

An **sma_orbit_position_and_velocity** specifies an orbit position and a velocity vector. This is an element of the ephemeris of an orbit arc. The position and velocity are defined in the coordinate_system of the applicable sma_space_mission_case. The applicable sma_space_mission_case is the one with the orbit_arc that references the **sma_orbit_position_and_velocity**.

Express specification:

```
ENTITY sma_orbit_position_and_velocity;
  position           : mgm_3d_cartesian_point;
  velocity_magnitude : nrf_real_quantity_value_prescription;
  velocity_direction : mgm_3d_direction;
WHERE
  wr1: velocity_magnitude.quantity_type.quantity_category.name = 'velocity';
  wr2: nrf_verify_dimensional_exponents(
 velocity_magnitude.quantity_type.quantity_category,
    1.0, 0.0, -1.0, 0.0, 0.0, 0.0, 0.0);
END_ENTITY;
```

Attribute definitions:

- position specifies the orbit position in the applicable space coordinate system.
- velocity_magnitude specifies the magnitude of the velocity vector.
- velocity_direction specifies the direction of the velocity vector.

Formal propositions:

- wr1: The name of the quantity_category of velocity_magnitude shall be 'velocity'.
- wr2: The dimension of velocity_magnitude shall be length over time.

4.5.4.5 ENTITY sma_discretized_orbit_arc

An **sma_discretized_orbit_arc** is a type of sma_orbit_arc that specifies a general orbit arc in the form of an ephemeris, i.e. as a list of positions with velocity vectors. This enables support of complex orbits, for example perturbed orbits. For informational purposes a set of Kepler orbit parameters may be referenced to indicate an associated unperturbed orbit arc.

Express specification:

```
ENTITY sma_discretized_orbit_arc
  SUBTYPE OF(sma_orbit_arc);
  orbit_generator_name               : nrf_label;
  positions_and_velocities           : LIST OF sma_orbit_position_and_velocity;
  centre_of_governing_celestial_body : OPTIONAL mgm_3d_cartesian_point;
  kepler_parameters                  : OPTIONAL sma_kepler_parameter_set;
END_ENTITY;
```


Attribute definitions:

— orbit_generator_name specifies the name of the generator that generated the orbit positions and velocities for the orbit arc.

— positions_and_velocities specifies the ephemeris of orbit positions and velocities that define the orbit arc.

— centre_of_governing_celestial_body optionally specifies the location of the centre of the celestial body that governs the orbit arc.

— kepler_parameters optionally specifies for informational purposes a set of Kepler orbit parameters that specify the undisturbed orbit arc.

4.5.4.6 ENTITY sma_kepler_parameter_set

An **sma_kepler_parameter_set** specifies the six classical Kepler parameters to specify an orbit arc: the semi major axis, the eccentricity, the inclination, the right ascension of the ascending node, the argument of periapsis and the true anomaly at the start of the arc. In addition also the true anomaly at the end of the arc is specifies to enable truncation of the arc.

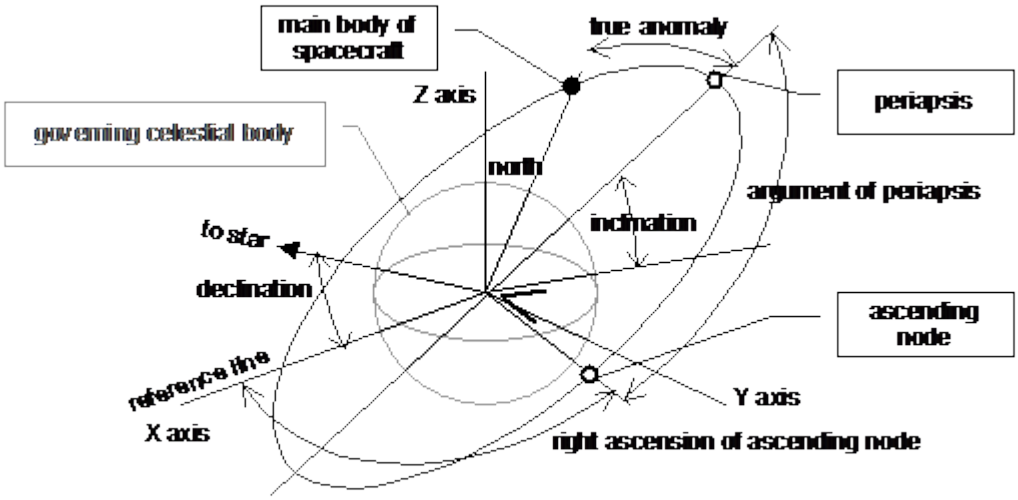

Figure 15 Illustration of parameters for a Keplerian elliptical orbit around a planet, in the typical "planet_centric_star_fixed_equatorial" space coordinate system

Express specification:

```
ENTITY sma_kepler_parameter_set;
  semi_major_axis                   : nrf_real_quantity_value_prescription;
  eccentricity                      : nrf_real_quantity_value_prescription;
  inclination                       : nrf_real_quantity_value_prescription;
  right_ascension_of_ascending_node : nrf_real_quantity_value_prescription;
  argument_of_periapsis             : nrf_real_quantity_value_prescription;
  true_anomaly_at_start             : nrf_real_quantity_value_prescription;
  true_anomaly_at_end               : nrf_real_quantity_value_prescription;
WHERE
  wr1: nrf_verify_dimensional_exponents(
 semi_major_axis.quantity_type.quantity_category,
    1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) AND
    (semi_major_axis.val > 0.0);
```



```
  wr2: nrf_verify_dimensional_exponents(
    eccentricity.quantity_type.quantity_category,
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) AND
    (eccentricity.val >= 0.0);
  wr3: (inclination.quantity_type.quantity_category.name = 'plane_angle') AND
    { -180.0 < inclination.val <= 180.0 };
  wr4: (right_ascension_of_ascending_node.quantity_type.quantity_category.name =
      'plane_angle') AND
    { -360.0 < right_ascension_of_ascending_node.val <= 360.0 };
  wr5: (argument_of_periapsis.quantity_type.quantity_category.name = 'plane_angle') AND
    { 0.0 <= argument_of_periapsis.val < 360.0 };
  wr6: (true_anomaly_at_start.quantity_type.quantity_category.name = 'plane_angle') AND
    { -360.0 < true_anomaly_at_start.val <= 360.0 };
  wr7: (true_anomaly_at_end.quantity_type.quantity_category.name = 'plane_angle') AND
    { true_anomaly_at_start.val <= true_anomaly_at_end.val
      <= true_anomaly_at_start.val + 360.0 };
END_ENTITY;
```

Attribute definitions:

- semi_major_axis specifies the length of the semi-major axis of the orbit arc.
- eccentricity specifies the eccentricity of the orbit arc.
- inclination specifies the inclination of the orbit arc with respect to the reference plane defined by the XY plane of the applicable space coordinate system.
- right_ascension_of_ascending_node specifies the right ascension angle of the ascending node of the orbit arc, i.e. the position where the orbit intersects with the reference plane defined by the XY plane of the applicable space coordinate system and the velocity vector points into the +Z half space.
- argument_of_periapsis specifies the true anomaly angle from the ascending node to the position of the periapsis of the orbit arc.
- true_anomaly_at_start specifies the true anomaly angle of the start position on the orbit arc measured from the periapsis position.
- true_anomaly_at_end specifies the true anomaly angle of the end position on the orbit arc measured from the periapsis position.

Formal propositions:

- wr1: The semi_major_axis shall be a positive length quantity type.
- wr2: The eccentricity shall be a non-dimensional non-negative real quantity type.
- wr3: The inclination shall be a 'plane_angle' in the interval from -180 degree exclusive to 180 degree inclusive.
- wr4: The right_ascension_of_ascending_node shall be a 'plane_angle' in the interval from -360 degree exclusive to 360 degree inclusive.
- wr5: The argument_of_periapsis shall be a 'plane_angle' in the interval from 0 degree inclusive to 360 degree exclusive.
- wr6: The true_anomaly_at_start shall be a 'plane_angle' in the interval from -360 degree exclusive to 360 degree inclusive.
- wr7: The true_anomaly_at_end shall be a 'plane_angle' in the interval from true_anomaly_at_start inclusive to true_anomaly_at_start plus 360 degree inclusive.

4.5.4.7 ENTITY sma_keplerian_orbit_arc

An **sma_keplerian_orbit_arc** is a type of orbit_arc that specifies an undisturbed basic Keplerian orbital arc.

Express specification:

```
ENTITY sma_keplerian_orbit_arc
  SUPERTYPE OF(ONEOF(
 sma_keplerian_orbit_arc_with_evaluation_interval,
 sma_keplerian_orbit_arc_with_evaluation_positions))
  SUBTYPE OF(sma_orbit_arc);
  kepler_parameters : sma_kepler_parameter_set;
END_ENTITY;
```

Attribute definitions:

— kepler_parameters specifies a set of Kepler orbit parameters that define the orbit arc.

4.5.4.8 ENTITY sma_keplerian_orbit_arc_with_evaluation_interval

An **sma_keplerian_orbit_arc_with_evaluation_interval** is a type of sma_keplerian_orbit_arc that specifies an undisturbed basic Keplerian orbital arc with an evaluation interval for analysis purposes.

Express specification:

```
ENTITY sma_keplerian_orbit_arc_with_evaluation_interval
  SUBTYPE OF(sma_keplerian_orbit_arc);
  evaluation_interval : nrf_real_quantity_value_prescription;
WHERE
  wr1: evaluation_interval.quantity_type.name IN
    ['true_anomaly_plane_angle', 'mission_elapsed_time', 'orbit_arc_length'];
END_ENTITY;
```

Attribute definitions:

— evaluation_interval specifies the interval for evaluation positions on the orbit arc through a constant increment in terms of true anomaly, mission elapsed time or distance along the orbit arc. From the value of evaluation_interval a minimum required set of evaluation positions are defined. This does not limit any implementation of an evaluation method to add additional evaluation positions, for example to represent eclipse entry or exit positions. The first evaluation position is by definition the start position defined by kepler_parameters.true_anomaly_at_start.

Formal propositions:

— wr1: The quantity type name of evaluation_interval shall be 'true_anomaly_plane_angle', 'mission_elapsed_time' or 'orbit_arc_length'.

4.5.4.9 ENTITY sma_keplerian_orbit_arc_with_evaluation_positions

An **sma_keplerian_orbit_arc_with_evaluation_positions** is a type of sma_keplerian_orbit_arc that specifies an undisturbed basic Keplerian orbital arc with a list of evaluation positions.

Express specification:


```
ENTITY sma_keplerian_orbit_arc_with_evaluation_positions
  SUBTYPE OF(sma_keplerian_orbit_arc);
  evaluation_positions : LIST [1:?] OF nrf_real_quantity_value_literal;
WHERE
  wr1: sma_verify_evaluation_positions(SELF);
END_ENTITY;
```

Attribute definitions:

— evaluation_positions specifies the list of evaluation positions on the orbit arc in terms of true anomaly, mission elapsed time or distance along the orbit arc.

Formal propositions:

— wr1: The quantity type name of the evaluation_positions shall be 'true_anomaly_plane_angle', 'mission_elapsed_time' or 'orbit_arc_length', and the values of subsequent positions shall be strictly increasing.

4.5.4.10 ENTITY sma_celestial_body_class

An **sma_celestial_body_class** is a specification of a class of sma_celestial_body instances, that is: a named category of celestial bodies which share common characteristics and behaviour.

EXAMPLE 4 - An example of an **sma_celestial_body_class** would be a class with name 'planets'.

Express specification:

```
ENTITY sma_celestial_body_class
  SUBTYPE OF(nrf_named_observable_item_class);
END_ENTITY;
```

4.5.4.11 ENTITY sma_celestial_body

An **sma_celestial_body** is a type of nrf_named_observable_item that specifies a celestial body by a basic set of attributes: id, name, description and mean_radius. Additional properties of a celestial body can be added through the use of nrf_datacube or nrf_quantity_value_prescription_for_item instances in the initializations or prescriptions attributes of an sma_space_environment.

EXAMPLE 5 - Examples of additional properties that may be added are the following real quantity types: 'solar_heat_flux', 'albedo_reflection', 'infra_red_radiation_temperature', 'planet_to_sun_notional_distance'.

Express specification:

```
ENTITY sma_celestial_body
  SUPERTYPE OF(ONEOF(sma_celestial_body_with_orbit))
  SUBTYPE OF(nrf_named_observable_item);
  SELF\nrf_named_observable_item.item_class : sma_celestial_body_class;
  mean_radius : nrf_real_quantity_value_prescription;
UNIQUE
  ur1: id;
  ur2: name;
WHERE
  wr1: nrf_verify_dimensional_exponents(
     mean_radius.quantity_type.quantity_category,
     1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```



```
   AND (mean_radius.val >= 0.0);
END_ENTITY;
```

Attribute definitions:

- item_class specifies the sma_celestial_body_class of which the **sma_celestial_body** is a member.
- mean_radius specifies the mean radius of the spheroid representing the celestial body.

Formal propositions:

- ur1: The id shall be unique in the dataset.
- ur2: The name shall be unique in the dataset.
- wr1: The mean_radius shall be a non-negative length quantity type.

4.5.4.12 ENTITY sma_celestial_body_with_orbit

An **sma_celestial_body_with_orbit** is a type of sma_celestial_body to which the specification of the orbit around its governing celestial body is added. This enables the definition of a complex orbital system by a chain of celestial bodies and orbits.

EXAMPLE 6 - Examples are a planet in its orbit around the Sun or the Moon in its orbit around Earth, and Earth in its orbit around the Sun.

Express specification:

```
ENTITY sma_celestial_body_with_orbit
  SUBTYPE OF(sma_celestial_body);
  orbit : sma_orbit_arc;
WHERE
  wr1: EXISTS(orbit.governing_celestial_body);
END_ENTITY;
```

Attribute definitions:

- orbit specifies the orbit of the celestial body around its governing_celestial_body.

Formal propositions:

- wr1: The orbit shall specify the governing_celestial_body.

4.5.4.13 ENTITY sma_space_environment

An **sma_space_environment** specifies the properties that represent the space environment for a space mission model and case.

Express specification:

```
ENTITY sma_space_environment;
  prescriptions    : LIST OF nrf_quantity_value_prescription_for_item;
  surface_material : OPTIONAL nrf_material;
  bulk_material    : OPTIONAL nrf_material;
INVERSE
```

```
  containing_case  : sma_space_mission_case FOR space_environment;
END_ENTITY;
```

Attribute definitions:

- prescriptions specifies the quantity value prescriptions for observable items that represent the space environment.
- surface_material optionally specifies the material that represents the surface of the space environment far away from the model. This may be a 'virtual' material that is only used in order to enable the specification of surface properties.
- bulk_material optionally specifies the material that represents the bulk of the space environment surrounding the model. This may be a 'virtual' material that is only used in order to enable the specification of bulk properties.
- containing_case inversely specifies the sma_space_mission_case that references this **sma_space_environment**.

4.5.4.14 ENTITY sma_kinematic_articulation

An **sma_kinematic_articulation** is an abstract supertype that provides a generic mechanism to reference an sma_parametric_kinematic_articulation, an sma_kinematic_articulation_with_pointing_constraint or an sma_fast_spinning_kinematic_articulation. It specifies an actual kinematic transformation (location and orientation) of a geometric item for which a kinematic joint is defined. This allows for the representation of moving rigid bodies.

Express specification:

```
ENTITY sma_kinematic_articulation
  ABSTRACT SUPERTYPE OF(ONEOF(
    sma_parametric_kinematic_articulation,
 sma_kinematic_articulation_with_pointing_constraint,
    sma_fast_spinning_kinematic_articulation));
  joint : skm_kinematic_joint;
END_ENTITY;
```

Attribute definitions:

- joint specifies the skm_kinematic_joint to which the kinematic articulation pertains.

4.5.4.15 ENTITY sma_parametric_kinematic_articulation

An **sma_parametric_kinematic_articulation** is a type of sma_kinematic_articulation that specifies kinematic movement by a collection of explicit sliding distance or rotation angle parameters. The number of parameters must correspond to the degrees of freedom of the associated kinematic joint. The kinematic articulation is applied after any possible static transformations have been applied to the affected geometric_item, as specified by the transformation attribute of an mgm_any_meshed_geometric_item.

Express specification:

```
ENTITY sma_parametric_kinematic_articulation
  SUBTYPE OF(sma_kinematic_articulation);
  parameters : LIST [1:6] OF nrf_real_quantity_value_prescription;
WHERE
  wr1: SIZEOF(parameters) = SIZEOF(joint.degrees_of_freedom);
```

```
  -- wr2: verify for each degree of freedom i:
  -- if sliding then parameters[i] is a 'length'
  -- if revolute then parameters[i] is a 'plane_angle'
END_ENTITY;
```

Attribute definitions:

- parameters specifies the list of articulation parameters for each of the degrees_of_freedom of the associated kinematic joint. An articulation parameter (i.e. an element in parameters) is a sliding distance for an skm_sliding_degree_of_freedom and a rotation angle for an skm_revolute_degree_of_freedom. Since the parameter values are specified through nrf_real_quantity_value_prescription instances any mathematical expression can be captured. Typical usage would be the rotation angle as a function of mission elapsed time or true anomaly.

Formal propositions:

- wr1: The number of parameters shall be equal to the number of degrees of freedom for the associated kinematic joint.
- wr2: TBD.

4.5.4.16 ENTITY sma_kinematic_articulation_with_pointing_constraint

An **sma_kinematic_articulation_with_pointing_constraint** is a type of sma_kinematic_articulation that specifies kinematic movement by a pointing constraint for a kinematic joint with one or two revolute degrees of freedom. For each revolute kinematic joint a pointer (which is a direction vector) is specified in the coordinate system of the affected geometric item. Also a desired pointing direction is specified. The pointing constraint requires then to orient the pointer of the geometric item such that the angle between the pointer and the desired pointing direction is minimized, in other words the pointer is aligned as much as possible with the desired pointing direction while still fullfilling possible additional constraints specified by rotation limit stops. This kinematic transformation is applied after any possible static transformations have been applied to the affected geometric_item, as specified by the transformation attribute of an mgm_any_meshed_geometric_item.

Express specification:

```
ENTITY sma_kinematic_articulation_with_pointing_constraint
  SUBTYPE OF(sma_kinematic_articulation);
  primary_constraint   : sma_kinematic_pointing_constraint;
  secondary_constraint : OPTIONAL sma_kinematic_pointing_constraint;
END_ENTITY;
```

Attribute definitions:

- primary_constraint specifies the primary pointing constraint, which specifies the pointer direction and the desired pointing direction.
- secondary_constraint optionally specifies the secondary pointing constraint, which specifies the pointer direction and the desired pointing direction.

4.5.4.17 ENTITY sma_kinematic_pointing_constraint

An **sma_kinematic_pointing_constraint** specifies a pointing constraint for a revolute kinematic joint by defining an axis of rotation and a pointer direction (both in the local coordinate system of the affected shape), and a desired pointing direction which can be parametric. Additionally rotation limit stops can be specified.

Express specification:

```
ENTITY sma_kinematic_pointing_constraint
  SUPERTYPE OF(ONEOF(
    sma_kinematic_cartesian_pointing_constraint,
    sma_kinematic_pointing_to_star_constraint,
 sma_kinematic_tracked_point_pointing_constraint));
  pointer          : mgm_3d_direction;
  desired_pointing : nrf_enumeration_quantity_value_literal;
WHERE
  wr1: sma_verify_kinematic_pointing_constraint(SELF);
END_ENTITY;
```

Attribute definitions:

- pointer specifies a direction vector with respect to the local coordinate system of the affected geometric item that defines the local direction that should be aligned as much as possible with the desired pointing direction under the pointing constraint.
- desired_pointing specifies the direction where the pointer should be pointing under the pointing constraint.

Formal propositions:

- wr1: The quantity type name of desired_pointing shall be 'pointing_in_space' and the possible desired pointing names shall include the list as defined in function sma_verify_kinematic_pointing_constraint.

4.5.4.18 ENTITY sma_kinematic_cartesian_pointing_constraint

An **sma_kinematic_cartesian_pointing_constraint** is a type of sma_kinematic_pointing_constraint that specifies a pointing constraint for a revolute kinematic joint through a general cartesian direction vector.

Express specification:

```
ENTITY sma_kinematic_cartesian_pointing_constraint
  SUBTYPE OF(sma_kinematic_pointing_constraint);
  cartesian_pointing : OPTIONAL mgm_3d_direction;
WHERE
  wr1: desired_pointing.quantity_type.enumeration_items[desired_pointing.val].name =
    'general';
END_ENTITY;
```

Attribute definitions:

- cartesian_pointing specifies a general pointing direction vector in the coordinate_system of the applicable sma_space_mission_case.

Formal propositions:

— wr1: The name of the desired_pointing shall be 'general'.

4.5.4.19 ENTITY sma_kinematic_pointing_to_star_constraint

An **sma_kinematic_pointing_to_star_constraint** is a type of sma_kinematic_pointing_constraint that specifies a pointing constraint for a revolute kinematic joint through a pointing direction to a star using a right ascension and a declination angle.

Express specification:

```
ENTITY sma_kinematic_pointing_to_star_constraint
  SUBTYPE OF(sma_kinematic_pointing_constraint);
  pointing_to_star : sma_pointing_to_star;
WHERE
  wr1: desired_pointing.quantity_type.enumeration_items[desired_pointing.val].name =
    'to_star';
END_ENTITY;
```

Attribute definitions:

— pointing_to_star specifies a pointing direction to a star in the coordinate_system of the applicable sma_space_mission_case.

Formal propositions:

— wr1: The name of the desired_pointing shall be 'to_star'.

4.5.4.20 ENTITY sma_kinematic_tracked_point_pointing_constraint

An **sma_kinematic_tracked_point_pointing_constraint** is a type of sma_kinematic_pointing_constraint that specifies a pointing constraint for a revolute kinematic joint by specifying a point to be tracked.

Express specification:

```
ENTITY sma_kinematic_tracked_point_pointing_constraint
  SUBTYPE OF(sma_kinematic_pointing_constraint);
  tracked_point : mgm_3d_cartesian_point;
WHERE
  wr1: desired_pointing.quantity_type.enumeration_items[desired_pointing.val].name =
    'tracked_point';
END_ENTITY;
```

Attribute definitions:

— tracked_point specifies the location of the point to be tracked in the coordinate_system of the applicable sma_space_mission_case. Since it may be specified as an mgm_parametric_3d_cartesian_point it can be a moving point, for instance as a function of mission elapsed time.

Formal propositions:

— wr1: The name of the desired_pointing shall be 'tracked_point'.

4.5.4.21 ENTITY sma_fast_spinning_kinematic_articulation

An **sma_fast_spinning_kinematic_articulation** is a type of sma_kinematic_articulation that specifies a fast spinning part. The purpose of this kinematic joint is to model a part that is continuously rotating at constant angular velocity. The spinning is 'fast' with respect to the typical (thermal) response time of the rotating part, hence for (thermal) analysis the computation procedure can be simplified by sampling a number of discrete spin positions and averaging the (thermal) analysis results.

Express specification:

```
ENTITY sma_fast_spinning_kinematic_articulation
  SUBTYPE OF(sma_kinematic_articulation);
  rotation_axis              : mgm_3d_direction;
  number_of_spin_positions   : nrf_positive_integer;
END_ENTITY;
```

Attribute definitions:

- rotation_axis specifies the axis about which the model is spinning.
- number_of_spin_positions specifies the number of discrete spin positions that shall be sampled for analysis purposes.

4.5.4.22 ENTITY sma_pointing_to_star

An **sma_pointing_to_star** specifies a pointing direction to a star through the spherical right ascension and spherical declination angles.

Express specification:

```
ENTITY sma_pointing_to_star;
  star                       : sma_celestial_body;
  spherical_right_ascension  : nrf_real_quantity_value_prescription;
  spherical_declination      : nrf_real_quantity_value_prescription;
WHERE
  wr1:
 (spherical_right_ascension.quantity_type.quantity_category.name = 'plane_angle')
    AND {-360.0 <= spherical_right_ascension.val <= 360.0};
  wr2:
 (spherical_declination.quantity_type.quantity_category.name = 'plane_angle')
    AND {-90.0 <= spherical_declination.val <= 90.0};
END_ENTITY;
```

Attribute definitions:

- star specifies the sma_celestial_body that represents a star.
- spherical_right_ascension specifies the spherical right ascension component of the direction toward the centre of the star. The angle is defined in the applicable space coordinate system. The right ascension is the angle in the XY plane from the X axis to the plane defined by the Z axis and the vector to the centre of the star
- spherical_declination specifies the spherical declination component of the direction toward the centre of the star. The angle is defined in the applicable space coordinate system. The declination is the angle from the XY plane in the plane defined by the Z axis and the vector to the centre of the star.

Formal propositions:

- wr1: The spherical_right_ascension is a 'plane_angle' quantity_type and its value is in the interval from -360 degree inclusive to 360 degree inclusive.
- wr1: The spherical_declination is a 'plane_angle' quantity_type and its value is in the interval from -90 degree inclusive to 90 degree inclusive.

4.5.4.23 FUNCTION sma_verify_kinematic_pointing_constraint

The function **sma_verify_kinematic_pointing_constraint** verifies that the quantity_type of the desired_pointing of an sma_kinematic_pointing_constraint is an enumeration quantity type with name 'pointing_in_space' and contains the following enumeration_items as a minimum:

- 'general'
- 'tracked_point'
- 'determined_by_next_lower_level_kinematic_articulation'
- 'sun'
- 'equatorial_projection_of_sun'
- 'vernal_equinox'
- 'ecliptic_north_pole'
- 'planet_north_pole'
- 'planet_nadir'
- 'planet_zenith'
- 'orbit_normal'
- 'anti_orbit_normal'
- 'spacecraft_velocity_vector'
- 'anti_spacecraft_velocity_vector'
- 'to_star'

The precise definitions of these pointings will be specified in the external STEP-TAS dictionary. The function returns TRUE if the constraints are met and FALSE otherwise.

Express specification:

```
FUNCTION sma_verify_kinematic_pointing_constraint(
  kpc : sma_kinematic_pointing_constraint) : BOOLEAN;
  LOCAL
    minimum_list_of_pointings : LIST OF STRING := [
      'general',
      'tracked_point',
 'determined_by_next_lower_level_kinematic_articulation',
      'sun',
      'equatorial_projection_of_sun',
      'vernal_equinox',
      'ecliptic_north_pole',
      'planet_north_pole',
      'planet_nadir',
      'planet_zenith',
      'orbit_normal',
      'anti_orbit_normal',
```



```
      'spacecraft_velocity_vector',
      'anti_spacecraft_velocity_vector',
      'to_star'];
    pointing_quantity_type : nrf_enumeration_quantity_type;
    found : BOOLEAN;
  END_LOCAL;
  pointing_quantity_type := kpc.desired_pointing.quantity_type;
  IF NOT (pointing_quantity_type.name = 'pointing_in_space') THEN
    RETURN(FALSE);
  END_IF;
  REPEAT i := 1 TO SIZEOF(minimum_list_of_pointings);
    found := FALSE;
    REPEAT j := 1 TO SIZEOF(pointing_quantity_type.enumeration_items) WHILE (NOT found);
      IF minimum_list_of_pointings[i] =
 pointing_quantity_type.enumeration_items[j].name THEN
        found := TRUE;
      END_IF;
    END_REPEAT;
    IF NOT found THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;

  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— kpc specifies the sma_kinematic_pointing_constraint to be verified.

4.5.4.24 FUNCTION sma_verify_kinematic_articulations

The function **sma_verify_kinematic_articulations** verifies that all elements defined in the sma_kinematic_articulation instances listed in the articulations of an sma_space_mission_case use the applicable context quantity types. The function returns TRUE if this is the case and FALSE otherwise.

Express specification:

```
FUNCTION sma_verify_kinematic_articulations(
    a_case : sma_space_mission_case) : BOOLEAN;
  LOCAL
    ka             : sma_kinematic_articulation;
  END_LOCAL;
  REPEAT i := 1 TO SIZEOF(a_case.articulations);
    ka := a_case.articulations[i];
    IF 'SMA_ARM.SMA_PARAMETRIC_KINEMATIC_ARTICULATION' IN TYPEOF(ka) THEN
      REPEAT j := 1 TO SIZEOF(ka.joint.degrees_of_freedom);
        IF 'SMA_ARM.SKM_SLIDING_DEGREE_OF_FREEDOM'
            IN TYPEOF(ka.joint.degrees_of_freedom[j]) THEN
          IF ka.parameters[j].quantity_type
              :<>: mgm_get_context_quantity_type(
              a_case.for_model, 'length') THEN
            RETURN(FALSE);
          END_IF;
        ELSE
          -- it is an mgm_rotation
          IF ka.parameters[j].quantity_type
              :<>: mgm_get_context_quantity_type(
              a_case.for_model, 'plane_angle') THEN
            RETURN(FALSE);
          END_IF;
```



```
        END_IF;
      END_REPEAT;
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— a_case specifies the sma_space_mission_case for which the articulations shall be verified.

4.5.4.25 FUNCTION sma_verify_evaluation_positions

The function **sma_verify_evaluation_positions** verifies that all evaluation_positions of an sma_keplerian_orbit_arc_with_evaluation_positions have a quantity_type with a name of 'true_anomaly_plane_angle', 'mission_elapsed_time' or 'orbit_arc_length' and that all positions use the same quantity_type. Furthermore the function verifies that the values of the subsequent positions are strictly increasing. The function returns TRUE if all contraints are fulfilled and FALSE otherwise.

Express specification:

```
FUNCTION sma_verify_evaluation_positions(
    koa : sma_keplerian_orbit_arc_with_evaluation_positions) : BOOLEAN;
  IF NOT (koa.evaluation_positions[1].quantity_type.name IN
      ['true_anomaly_plane_angle', 'mission_elapsed_time', 'orbit_arc_length']) THEN
    RETURN(FALSE);
  END_IF;
  REPEAT i := 2 TO SIZEOF(koa.evaluation_positions);
    IF NOT(koa.evaluation_positions[i].quantity_type :=:
        koa.evaluation_positions[1].quantity_type) THEN
      RETURN(FALSE);
    END_IF;
    IF NOT(koa.evaluation_positions[i].val >
        koa.evaluation_positions[i-1].val) THEN
      RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
```

Argument definitions:

— koa specifies the sma_keplerian_orbit_arc_with_evaluation_positions for which the evaluation_positions shall be verified.

4.5.5 END_SCHEMA declaration for sma_arm

The following EXPRESS declaration ends the sma_arm schema.

Express specification:

```
END_SCHEMA; -- sma_arm
```

4.6 STEP-TAS procotol

4.6.1 SCHEMA declaration for tas_arm

The following EXPRESS declaration begins the tas_arm schema.

Express specification:

```
SCHEMA tas_arm;
-- $Id$
-- Copyright (c) 1995-2018 European Space Agency (ESA)
-- All rights reserved.
```

4.6.2 Interfaced schema(ta) for tas_arm

The STEP-TAS protocol consists of the four modules described above: nrf_arm schema as specified in [STEP-NRF], mgm_arm, skm_arm and sma_arm. This implies that so-called *long form* schema of the tas_arm protocol is the full expansion of the four module schemata.

Express specification:

```
USE FROM nrf_arm;
USE FROM mgm_arm;
USE FROM skm_arm;
USE FROM sma_arm;
```

4.6.3 CONSTANT specifications

There is one constant defined for the TAS protocol: SCHEMA_OBJECT_IDENTIFIER.

Express specification:

```
CONSTANT
  SCHEMA_OBJECT_IDENTIFIER : STRING :=
  '{http://www.purl.org/ESA/step-tas/v6.0/tas_arm.exp}';
  -- in formal version to be replaced with
  -- '{ iso standard n part(p) version(v) }'
END_CONSTANT;
```

Constant definitions:

SCHEMA_OBJECT_IDENTIFIER provides a built-in way to reference the object identifier of the protocol for version verification. For the definition and usage of the object identifier see ISO 10303-1 and Annex E.

4.6.4 END_SCHEMA declaration for tas_arm

The following EXPRESS declaration ends the tas_arm schema.

Express specification:

```
END_SCHEMA; -- tas_arm
```

5 Conformance requirements

5.1 Conformance requirements for STEP-TAS as a whole

Conformance to this application protocol includes satisfying the requirements stated in this part, the requirements of the implementation method(s) supported and the relevant requirements of the normative references.

An implementation shall support at least one of the following implementation methods:

— ISO 10303-21.

Requirements with respect to implementation methods specific requirements are specified in Annex C.

The Protocol Implementation Conformance Statement (PICS) proforma lists the options or the combinations of options that may be included in the implementation. The PICS proforma is provided in Annex D.

This application protocol provides for a number of options that may be supported by an implementation. These options have been grouped into the following conformance classes:

— CC1: Thermal radiation and conduction model defined by shell geometry.
— CC2: CC1 plus kinematic model.
— CC3: CC1 plus constructive geometry.
— CC4: CC3 plus kinematic model.
— CC5: CC1 plus space mission aspects.
— CC6: CC4 plus space mission aspects.
— CC7: Results for thermal radiation and conduction model.
— CC8: Thermal lumped parameter model without user-defined logic.
— CC9: CC8 plus results.
— CC10: Thermal lumped parameter model with user-defined logic.
— CC11: CC10 plus results;
— CC12: Thermal test or operation model with results.

Conformance to a particular class requires that all protocol elements defined as part of that class be supported. The following table defines for each protocol element in which classes it shall be supported: *Y* for *Yes, shall be supported*, *N* for *No, does not need to be supported*.

Table 4 – Conformance class elements

protocol element	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11	CC12
nrf_address	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_algorithmic_language	N	N	N	N	N	N	N	N	N	Y	Y	N
nrf_anti_symmetric_matrix_quantity_type	N	N	N	N	N	N	N	Y	Y	Y	Y	N
nrf_approval	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_calendar_date	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_case	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

protocol element	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11	CC12
nrf_case_event	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_case_interval	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_context_dependent_unit	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_conversion_based_unit	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_coordinated_universal_time_offset	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_cyclic_real_interpolation_table_expression	N	N	N	N	N	N	N	Y	Y	Y	Y	N
nrf_datacube	N	N	N	N	N	N	Y	N	Y	N	Y	Y
nrf_datacube_derivation_relationship	N	N	N	N	N	N	Y	N	Y	N	Y	Y
nrf_date_and_time	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_derivation_procedure	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y
nrf_derived_unit	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_derived_unit_element	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_enumeration_item	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_enumeration_quantity_type	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_enumeration_quantity_value_expression	N	N	N	N	N	N	N	N	N	Y	Y	N
nrf_enumeration_quantity_value_literal	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_extended_si_unit	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_formal_parameter	N	N	N	N	N	N	N	N	N	Y	Y	N
nrf_general_tensor_quantity_type	N	N	N	N	N	N	N	Y	Y	Y	Y	N
nrf_integer_quantity_type	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_integer_quantity_value_expression	N	N	N	N	N	N	N	N	N	Y	Y	N
nrf_integer_quantity_value_literal	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_local_time	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_material	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_material_class	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_model_constraint	N	N	N	N	N	N	N	N	N	Y	Y	N
nrf_model_function	N	N	N	N	N	N	N	N	N	Y	Y	N
nrf_model_represents_product_relationship	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_named_observable_item	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_named_observable_item_class	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_named_observable_item_group	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_named_observable_item_group_class	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_named_observable_item_list	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_network_model	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_network_model_class	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_network_model_nodes_mapping	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_network_node	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_network_node_class	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_network_node_mapping	N	N	N	N	N	N	N	Y	Y	Y	Y	Y
nrf_network_node_relationship	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_network_node_relationship_class	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_observable_item_list	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_organization	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_organizational_address	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_organizational_project	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_person	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_person_and_organization	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_personal_address	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_primary_physical_quantity_category	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_product	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

protocol element	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11	CC12
nrf_product_context	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_product_definition	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_product_definition_context	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_product_next_assembly_usage_relationship	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_product_version	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_qualified_physical_quantity_category	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_quantity_qualifier	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_quantity_type_list	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_quantity_value_prescription_for_item	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_real_interpolation_table_expression	N	N	N	N	N	N	N	Y	Y	Y	Y	N
nrf_real_lookup_table	N	N	N	N	N	N	N	Y	Y	Y	Y	N
nrf_real_quantity_type	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_real_quantity_value_expression	N	N	N	N	N	N	N	N	N	Y	Y	N
nrf_real_quantity_value_literal	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_real_univariate_power_series_polynomial_expression	N	N	N	N	N	N	N	Y	Y	Y	Y	N
nrf_root	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_run	N	N	N	N	N	N	Y	N	Y	N	Y	Y
nrf_secondary_physical_quantity_category	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_security_classification_level	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_state_list	N	N	N	N	N	N	Y	N	Y	N	Y	Y
nrf_string_quantity_type	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_string_quantity_value_expression	N	N	N	N	N	N	N	N	N	N	Y	N
nrf_string_quantity_value_literal	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_symmetric_matrix_quantity_type	N	N	N	N	N	N	N	N	N	N	Y	N
nrf_tensor_characteristic	N	N	N	N	N	N	N	N	N	N	Y	N
nrf_tensor_element	N	N	N	N	N	N	N	N	N	N	Y	N
nrf_tensor_quantity_value_expression	N	N	N	N	N	N	N	N	N	N	Y	N
nrf_tensor_quantity_value_literal	N	N	N	N	N	N	N	N	N	N	Y	N
nrf_tool_or_facility	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
nrf_uncertainty_probability_distribution	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y
nrf_uncertainty_specification_method	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y
nrf_variable	N	N	N	N	N	N	N	Y	Y	Y	Y	Y
mgm_3d_cartesian_point	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_3d_direction	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_axis_placement	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_axis_transformation_sequence	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_colour_rgb	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_compound_meshed_geometric_item	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_cone	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_cylinder	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_disc	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_enclosure	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_face	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_face_pair	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_half_space_solid	N	N	Y	Y	N	Y	Y	N	N	N	N	N
mgm_infinite_solid_by_plane	N	N	Y	Y	N	Y	Y	N	N	N	N	N
mgm_infinite_solid_cylinder	N	N	Y	Y	N	Y	Y	N	N	N	N	N
mgm_meshed_boolean_difference_surface	N	N	Y	Y	N	Y	Y	N	N	N	N	N
mgm_meshed_geometric_item_by_submodel	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_meshed_geometric_model	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N

protocol element	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11	CC12
mgm_meshed_primitive_bounded_surface	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_paraboloid	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_parametric_3d_cartesian_point	N	N	N	N	N	N	N	N	N	N	N	N
mgm_parametric_3d_direction	N	N	N	N	N	N	N	N	N	N	N	N
mgm_parametric_rotation_with_axes_fixed	N	N	N	N	N	N	N	N	N	N	N	N
mgm_parametric_rotation_with_axes_moving	N	N	N	N	N	N	N	N	N	N	N	N
mgm_parametric_translation	N	N	N	N	N	N	N	N	N	N	N	N
mgm_quadrilateral	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_qualified_compound_meshed_primitive_bounded_surface	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_quantity_context	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_rectangle	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_rotation_with_axes_fixed	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_rotation_with_axes_moving	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_solid_box	N	N	Y	Y	N	Y	Y	N	N	N	N	N
mgm_solid_cone	N	N	Y	Y	N	Y	Y	N	N	N	N	N
mgm_solid_cylinder	N	N	Y	Y	N	Y	Y	N	N	N	N	N
mgm_solid_paraboloid	N	N	Y	Y	N	Y	Y	N	N	N	N	N
mgm_solid_sphere	N	N	Y	Y	N	Y	Y	N	N	N	N	N
mgm_solid_triangular_prism	N	N	Y	Y	N	Y	Y	N	N	N	N	N
mgm_sphere	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_translation	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
mgm_triangle	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
skm_kinematic_degree_of_freedom	N	Y	N	Y	N	Y	Y	N	N	N	N	N
skm_kinematic_joint	N	Y	N	Y	N	Y	Y	N	N	N	N	N
skm_revolute_degree_of_freedom	N	Y	N	Y	N	Y	Y	N	N	N	N	N
skm_sliding_degree_of_freedom	N	Y	N	Y	N	Y	Y	N	N	N	N	N
sma_celestial_body	N	N	N	N	Y	Y	Y	N	N	N	N	N
sma_celestial_body_class	N	N	N	N	Y	Y	Y	N	N	N	N	N
sma_celestial_body_with_orbit	N	N	N	N	Y	Y	Y	N	N	N	N	N
sma_discretized_orbit_arc	N	N	N	N	Y	Y	Y	N	N	N	N	N
sma_fast_spinning_kinematic_articulation	N	N	N	N	N	Y	Y	N	N	N	N	N
sma_kepler_parameter_set	N	N	N	N	Y	Y	Y	N	N	N	N	N
sma_keplerian_orbit_arc	N	N	N	N	Y	Y	Y	N	N	N	N	N
sma_keplerian_orbit_arc_with_evaluation_interval	N	N	N	N	Y	Y	Y	N	N	N	N	N
sma_keplerian_orbit_arc_with_evaluation_positions	N	N	N	N	Y	Y	Y	N	N	N	N	N
sma_kinematic_articulation_with_pointing_constraint	N	N	N	N	N	Y	Y	N	N	N	N	N
sma_kinematic_cartesian_pointing_constraint	N	N	N	N	N	Y	Y	N	N	N	N	N
sma_kinematic_pointing_constraint	N	N	N	N	N	Y	Y	N	N	N	N	N
sma_kinematic_pointing_to_star_constraint	N	N	N	N	N	Y	Y	N	N	N	N	N
sma_kinematic_tracked_point_pointing_constraint	N	N	N	N	N	Y	Y	N	N	N	N	N
sma_orbit_position_and_velocity	N	N	N	N	Y	Y	Y	N	N	N	N	N
sma_parametric_kinematic_articulation	N	N	N	N	N	Y	Y	N	N	N	N	N
sma_pointing_to_star	N	N	N	N	N	Y	Y	N	N	N	N	N
sma_space_coordinate_system	N	N	N	N	Y	Y	Y	N	N	N	N	N
sma_space_environment	N	N	N	N	Y	Y	Y	N	N	N	N	N
sma_space_mission_case	N	N	N	N	Y	Y	Y	N	N	N	N	N

5.2 Conformance requirements for STEP-NRF in isolation

Conformance to this STEP-NRF protocol in isolation, or to an application protocol other than STEP-TAS that uses it, includes satisfying the requirements stated in this part, the requirements of the implementation method(s) supported and the relevant requirements of the normative references.

An implementation shall support at least one of the following implementation methods:

— ISO 10303-21.

Requirements with respect to implementation methods specific requirements are specified in Annex C.

The Protocol Implementation Conformance Statement (PICS) proforma lists the options or the combinations of options that may be included in the implementation. The PICS proforma is provided in Annex D.

This application protocol provides for two options that may be supported by an implementation. These options have been grouped into the following conformance classes:

— NRF-CC1: Network-models and results without parseable expressions and algorithms.
— NRF-CC2: Network-models and results with parseable expressions and algorithms.

Note that all of the STEP-TAS conformance classes satisfy NRF-CC1 except for CC10 and CC11 which satisfy NRF-CC2.

Conformance to a particular class requires that all protocol elements defined as part of that class be supported. The following table defines for each protocol element in which classes it shall be supported: *Y* for *Yes, shall be supported*, *N* for *No, does not need to be supported*.

Table 3: Conformance class elements

protocol element	NRF-CC1	NRF-CC2
nrf_address	Y	Y
nrf_algorithmic_language	N	Y
nrf_anti_symmetric_matrix_quantity_type	Y	Y
nrf_approval	Y	Y
nrf_calendar_date	Y	Y
nrf_case	Y	Y
nrf_case_event	Y	Y
nrf_case_interval	Y	Y
nrf_context_dependent_unit	Y	Y
nrf_conversion_based_unit	Y	Y
nrf_coordinated_universal_time_offset	Y	Y
nrf_cyclic_real_interpolation_table_expression	Y	Y
nrf_datacube	Y	Y
nrf_datacube_derivation_relationship	Y	Y
nrf_date_and_time	Y	Y
nrf_derivation_procedure	Y	Y
nrf_derived_unit	Y	Y
nrf_derived_unit_element	Y	Y
nrf_enumeration_item	Y	Y
nrf_enumeration_quantity_type	Y	Y
nrf_enumeration_quantity_value_expression	N	Y

protocol element	NRF-CC1	NRF-CC2
nrf_enumeration_quantity_value_literal	Y	Y
nrf_extended_si_unit	Y	Y
nrf_formal_parameter	N	Y
nrf_general_tensor_quantity_type	Y	Y
nrf_integer_quantity_type	Y	Y
nrf_integer_quantity_value_expression	N	Y
nrf_integer_quantity_value_literal	Y	Y
nrf_local_time	Y	Y
nrf_material	Y	Y
nrf_material_class	Y	Y
nrf_model_constraint	N	Y
nrf_model_function	N	Y
nrf_model_represents_product_relationship	Y	Y
nrf_named_observable_item	Y	Y
nrf_named_observable_item_class	Y	Y
nrf_named_observable_item_group	Y	Y
nrf_named_observable_item_group_class	Y	Y
nrf_named_observable_item_list	Y	Y
nrf_network_model	Y	Y
nrf_network_model_class	Y	Y
nrf_network_model_nodes_mapping	Y	Y
nrf_network_node	Y	Y
nrf_network_node_class	Y	Y
nrf_network_node_mapping	Y	Y
nrf_network_node_relationship	Y	Y
nrf_network_node_relationship_class	Y	Y
nrf_observable_item_list	Y	Y
nrf_organization	Y	Y
nrf_organizational_address	Y	Y
nrf_organizational_project	Y	Y
nrf_person	Y	Y
nrf_person_and_organization	Y	Y
nrf_personal_address	Y	Y
nrf_primary_physical_quantity_category	Y	Y
nrf_product	Y	Y
nrf_product_context	Y	Y
nrf_product_definition	Y	Y
nrf_product_definition_context	Y	Y
nrf_product_next_assembly_usage_relationship	Y	Y
nrf_product_version	Y	Y
nrf_qualified_physical_quantity_category	Y	Y
nrf_quantity_qualifier	Y	Y
nrf_quantity_type_list	Y	Y
nrf_quantity_value_prescription_for_item	Y	Y
nrf_real_interpolation_table_expression	Y	Y
nrf_real_lookup_table	Y	Y
nrf_real_quantity_type	Y	Y
nrf_real_quantity_value_expression	N	Y
nrf_real_quantity_value_literal	Y	Y
nrf_real_univariate_power_series_polynomial_expression	Y	Y

protocol element	NRF-CC1	NRF-CC2
nrf_root	Y	Y
nrf_run	Y	Y
nrf_secondary_physical_quantity_category	Y	Y
nrf_security_classification_level	Y	Y
nrf_state_list	Y	Y
nrf_string_quantity_type	Y	Y
nrf_string_quantity_value_expression	N	Y
nrf_string_quantity_value_literal	Y	Y
nrf_symmetric_matrix_quantity_type	Y	Y
nrf_tensor_characteristic	Y	Y
nrf_tensor_element	Y	Y
nrf_tensor_quantity_value_expression	N	Y
nrf_tensor_quantity_value_literal	Y	Y
nrf_tool_or_facility	Y	Y
nrf_uncertainty_probability_distribution	Y	Y
nrf_uncertainty_specification_method	Y	Y
nrf_variable	Y	Y

Annex A (normative) ARM EXPRESS expanded listing

Downloadable from https://exchange.esa.int/restricted/ecss-e-st-31-04/annex-a.zip

Annex B (informative) Application protocol usage guide

B.1 Dictionary of standard pre-defined entities

The STEP-TAS and STEP-NRF protocols are designed to be used in conjunction with a dictionary of standard pre-defined entities which describe and extend the particular problem domain. The protocols themselves are intended to describe relatively abstract and generic concepts which are likely to remain unchanged over time therefore avoiding unnecessary cycles of protocol update and revalidation. The dictionary may be extended over time independently of the protocol.

The dictionary defines basic additional entities, such as units (e.g. metre, kilogram) and quantity types and their associated units (e.g. mass, length) as well as domain specific quantities and enumerations. Such enumerations can be used to differentiate between classes of entities derived from their abstract STEP-TAS and STEP-NRF counterparts.

All applications that use a particular dictionary have the advantage that they share common names and definitions for domain-specific entities and therefore significantly reduce the risk of implementation incompatibilies when exchanging models.

B.2 Use of the STEP-TAS dictionary

An application that implements the STEP-TAS standard shall use the STEP-TAS *dictionary* and shall have the capability to load it at runtime. This dictionary is an ISO 10303-21 file ("STEP file") which itself conforms to the tas_arm SCHEMA.

The most recent STEP-TAS dictionary can be downloaded from:
http://www.purl.org/ESA/step-tas/v6.0/dictionary/tas_arm_dictionary.stp

B.3 Use of the STEP-NRF dictionary

NOTE: There is currently no standalone STEP-NRF-only dictionary because STEP-NRF is an abstract protocol designed to be a building block for more domain specific protocols with their own dictionaries, such as STEP-TAS.

Standards using STEP-NRF may make use of a *dictionary* in conjunction with the nrf_arm SCHEMA or a SCHEMA in which nrf_arm is used. The dictionary is an ISO 10303-21 file ("STEP file") which itself conforms to the nrf_arm SCHEMA or a SCHEMA in which nrf_arm is used. In this dictionary a number of instances are predefined. An application then loads the dictionary at runtime when it starts to create a dataset.

Instances of the following application objects can be defined in a STEP-NRF dictionary:

- nrf_calendar_date
- nrf_context_dependent_unit
- nrf_conversion_based_unit

- nrf_coordinated_universal_time_offset
- nrf_date_and_time
- nrf_derived_unit
- nrf_derived_unit_element
- nrf_enumeration_item
- nrf_enumeration_quantity_type
- nrf_local_time
- nrf_material_class
- nrf_network_model_class
- nrf_network_node_class
- nrf_network_node_relationship_class
- nrf_organization
- nrf_organizational_address
- nrf_organizational_project
- nrf_person
- nrf_person_and_organization
- nrf_quantity_qualifier
- nrf_real_quantity_type
- nrf_root
- nrf_scalar_quantity_category
- nrf_extended_si_unit

!!TBD!! List to be completed...

Annex C (informative) Bibliography

1. Article explaining physical quantities and units of measurement, including the International System of Units (SI):
   http://en.wikipedia.org/wiki/Units_of_measurement
   The URL was correct at the time of publication of this standard.
2. Article explaining the Internation System of Units (SI), ISO 31:
   http://en.wikipedia.org/wiki/International_System_of_Units
   The URL was correct at the time of publication of this standard.
3. Article explaining prefixes for binary data units conforming to IEC 60027-2:
   http://en.wikipedia.org/wiki/Binary_prefix
   The URL was correct at the time of publication of this standard.
4. NIST Technical Note 1297 - 1994 Edition - Guidelines for Evaluating and Expressing the Uncertainty of NIST Measurement Results
   http://www.physics.nist.gov/Pubs/guidelines/contents.html
   The URL was correct at the time of publication of this standard.

