

International Telecommunication Union

**ITU-T**

TELECOMMUNICATION  
STANDARDIZATION SECTOR  
OF ITU

**Z.151**

(10/2018)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE  
ASPECTS FOR TELECOMMUNICATION SYSTEMS

Formal description techniques (FDT) – User Requirements  
Notation (URN)

---

## **User Requirements Notation (URN) – Language definition**

Recommendation ITU-T Z.151

ITU-T



ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

FORMAL DESCRIPTION TECHNIQUES (FDT)	
Specification and Description Language (SDL)	Z.100–Z.109
Application of formal description techniques	Z.110–Z.119
Message Sequence Chart (MSC)	Z.120–Z.129
<b>User Requirements Notation (URN)</b>	<b>Z.150–Z.159</b>
Testing and Test Control Notation (TTCN)	Z.160–Z.179
PROGRAMMING LANGUAGES	
CHILL: The ITU-T high level language	Z.200–Z.209
MAN-MACHINE LANGUAGE	
General principles	Z.300–Z.309
Basic syntax and dialogue procedures	Z.310–Z.319
Extended MML for visual display terminals	Z.320–Z.329
Specification of the man-machine interface	Z.330–Z.349
Data-oriented human-machine interfaces	Z.350–Z.359
Human-machine interfaces for the management of telecommunications networks	Z.360–Z.379
QUALITY	
Quality of telecommunication software	Z.400–Z.409
Quality aspects of protocol-related Recommendations	Z.450–Z.459
METHODS	
Methods for validation and testing	Z.500–Z.519
MIDDLEWARE	
Processing environment architectures	Z.600–Z.609

*For further details, please refer to the list of ITU-T Recommendations.*

# Recommendation ITU-T Z.151

## User Requirements Notation (URN) – Language definition

### Summary

Recommendation ITU-T Z.151 defines the User Requirements Notation (URN) intended for the elicitation, analysis, specification and validation of requirements. URN combines modelling concepts and notations for goals (mainly for non-functional requirements and quality attributes) and scenarios (mainly for operational requirements, functional requirements and performance and architectural reasoning). The goal sub-notation is called Goal-oriented Requirements Language (GRL) and the scenario sub-notation is called Use Case Map (UCM).

### History

Edition	Recommendation	Approval	Study Group	Unique ID*
1.0	ITU-T Z.151	2008-11-13	17	<a href="http://handle.itu.int/11.1002/1000/9623">11.1002/1000/9623</a>
1.1	ITU-T Z.151 (2008) Cor. 1	2012-04-29	17	<a href="http://handle.itu.int/11.1002/1000/11584">11.1002/1000/11584</a>
2.0	ITU-T Z.151	2012-10-14	17	<a href="http://handle.itu.int/11.1002/1000/11760">11.1002/1000/11760</a>
3.0	ITU-T Z.151	2018-10-14	17	<a href="http://handle.itu.int/11.1002/1000/13711">11.1002/1000/13711</a>

---

\* To access the Recommendation, type the URL <http://handle.itu.int/> in the address field of your web browser, followed by the Recommendation's unique ID. For example, <http://handle.itu.int/11.1002/1000/11830-en>.

## FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications, information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

## NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

## INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at <http://www.itu.int/ITU-T/ipr/>.

© ITU 2019

All rights reserved. No part of this publication may be reproduced, by any means whatsoever, without the prior written permission of ITU.

## Table of Contents

	<b>Page</b>
1	Scope..... 1
1.1	Goal modelling with URN..... 1
1.2	Scenario modelling with URN..... 2
1.3	Documentation structure..... 3
2	References..... 3
3	Definitions..... 4
3.1	Terms defined elsewhere..... 4
3.2	Terms defined in this Recommendation..... 4
4	Abbreviations and acronyms..... 5
5	Conventions..... 6
5.1	Grammars..... 6
5.2	Basic definitions..... 6
5.3	Presentation style..... 6
6	URN basic structural features..... 7
6.1	URN abstract grammar metaclasses..... 8
6.2	URN concrete grammar metaclasses..... 12
7	GRL features..... 14
7.1	GRL basic structural features..... 15
7.2	GRL actors..... 19
7.3	GRL intentional elements..... 21
7.4	GRL links..... 24
7.5	GRL strategies..... 32
7.6	GRL indicators..... 37
7.7	GRL contribution contexts..... 43
7.8	GRL concrete grammar metaclasses..... 46
8	UCM features..... 62
8.1	UCM basic structural features..... 63
8.2	UCM maps and path nodes..... 65
8.3	UCM stubs and plug-ins..... 92
8.4	UCM components..... 102
8.5	UCM scenario definitions..... 110
8.6	UCM performance annotations..... 120
8.7	UCM concrete grammar metaclasses..... 130
9	Data language..... 131
9.1	URN data model..... 131
9.2	URN data types..... 132
9.3	Grammar for expressions..... 134
9.4	Grammar for actions..... 135
9.5	Grammar for failures..... 136

	<b>Page</b>
10 URN interchange format .....	137
11 URN analysis .....	137
11.1 GRL model evaluation.....	138
11.2 UCM scenario path traversal .....	140
12 Compliance statement .....	144
13 Tool compliance.....	149
13.1 Definitions of valid tools .....	149
13.2 Conformance.....	150
Annex A – URN interchange format: XML schema .....	159
Annex B – Textual URN specification.....	178
B.1 Introduction to the Textual URN specification.....	170
B.2 User Requirements Notation: core concepts.....	171
B.3 Goal-oriented Requirement Language.....	175
B.4 Use Case Map.....	182
Appendix I – Summary of the URN.....	212
I.1 Summary of abstract metamodel .....	204
I.2 Summary of concrete metamodel .....	209
I.3 Summary of URN symbols.....	212
Appendix II – Examples of GRL model evaluation algorithms .....	221
II.1 Introduction.....	214
II.2 Example of quantitative evaluation algorithm.....	216
II.3 Example of qualitative evaluation algorithm.....	220
II.4 Example of hybrid evaluation algorithm .....	227
II.5 Calculating with exceeding expectations.....	228
Appendix III – Examples of UCM path traversal mechanisms .....	236
III.1 Introduction.....	229
III.2 Example of depth-first UCM path traversal mechanism .....	229
III.3 Example of breadth-first UCM path traversal mechanism .....	234
Bibliography .....	241

## **Introduction**

### **Coverage**

User Requirement Notation (URN) has concepts for the specification of goals, non-functional requirements, rationales, indicators, behaviour, scenarios and structuring. This Recommendation focuses on the definition of an abstract syntax, a concrete graphical syntax and an interchange format for URN. An assessment of conformity of the current URN representation to the language requirements for URN (Recommendation ITU-T Z.150) is also included.

### **Application**

URN is applicable within standards bodies and industry. URN helps to describe and communicate requirements, and to develop reasoning about them. The main application areas include telecommunication systems, services and business processes, but URN is generally suitable for describing most types of reactive systems and information systems. The range of applications is from descriptions of business goals and requirements to high-level system design and architecture.

### **Status/Stability**

This Recommendation contains the stable definition of URN. URN components for goal modelling and scenario modelling have been used for more than a decade. The main text is accompanied by the following:

- Annex A: URN Interchange Format: XML Schema
- Annex B: Textual URN Specification
- Appendix I: Summary of the URN
- Appendix II: Examples of GRL Model Evaluation Algorithms
- Appendix III: Examples of UCM Path Traversal Mechanisms
- URN Change Request Form



# Recommendation ITU-T Z.151

## User Requirements Notation (URN) – Language definition

### 1 Scope

This Recommendation defines the User Requirements Notation (URN) intended for the elicitation, analysis, specification and validation of requirements. URN allows software and requirements engineers to discover and specify requirements for a proposed system or an evolving system, and analyse such requirements for correctness and completeness.

URN combines modelling concepts and notations for goals and intentions (mainly for non-functional requirements and quality attributes) and scenarios (mainly for operational requirements, functional requirements and performance and architectural reasoning). In particular, URN has concepts for the specification of goals, non-functional requirements, rationales, indicators, behaviour, scenarios and structuring.

This Recommendation focuses on the definition of an abstract syntax, a concrete graphical syntax, and an interchange format for URN. An assessment of conformity of the current URN representation to the language requirements for URN [ITU-T Z.150] is also included.

URN is applicable within standards bodies and industry. URN helps to describe and communicate requirements, and to develop reasoning about them. The main application areas include telecommunications systems, services and business processes, but URN is generally suitable for describing most types of reactive systems and information systems. The range of applications is from business descriptions of goals and requirements to high-level design.

URN is a notation that complies with [ITU-T Z.150]. It includes concepts and notations satisfying the language requirements of Z.150's URN-NFR (for non-functional requirements) and URN-FR (for functional requirements). URN integrates these concepts and notation into a single language.

#### 1.1 Goal modelling with URN

The subset of the URN language that addresses ITU-T Z.150 URN-NFR language requirements is named Goal-oriented Requirement Language (GRL), which is a language for supporting goal-oriented modelling and reasoning about requirements, especially non-functional requirements and quality attributes. It provides constructs for expressing various types of concepts that appear during the requirement process. GRL has its roots in two widespread goal-oriented modelling languages: *i\** and the NFR Framework. Major benefits of GRL over other popular notations include the integration of GRL with a scenario notation and a clear separation of GRL model elements from their graphical representation, enabling a scalable and consistent representation of multiple views/diagrams of the same goal model.

There are four main categories of concepts in GRL: actors, intentional elements, indicators and links. The intentional elements in GRL are goals, softgoals, tasks, resources and beliefs. They are intentional because they are used for models that allow answering questions such as why particular behaviours, and informational and structural aspects were chosen to be included in the system requirements, what alternatives were considered, what criteria were used to deliberate among alternative options, and what the reasons were for choosing one alternative over the other. Actors are holders of intentions; they are the active entities in the system or its environment (e.g., stakeholders or other systems) who want goals to be achieved, tasks to be performed, resources to be available and softgoals to be satisfied. Indicators make real-world measurements available for reasoning in the goal model, allowing for a more accurate assessment of the satisfaction of actors. Links are used to connect isolated elements in the requirement model. Different types of links depict different structural and intentional relationships (including decompositions, contributions and dependencies).

This kind of modelling is different from the detailed specification of "what" is to be done. Here the modeller is primarily concerned with exposing "why" certain choices for behaviour and/or structure were made or constraints introduced. The modeller is not yet interested in the operational details of processes or system requirements, or component interactions. Omitting these kinds of details during early development and standardization phases allows taking a higher level (sometimes called a strategic stance) towards modelling the current or the future standard or software system and its embedding environment. Modelling and answering "why" questions leads one to consider the opportunities stakeholders seek out and/or vulnerabilities they try to avoid within their environment by utilizing capabilities of the software system and/or other stakeholders, by trying to rely upon and/or assign capabilities and by introducing constraints on how those capabilities ought to be performed.

GRL supports the analysis of strategies, which help reach the most appropriate trade-offs among (often conflicting) goals of stakeholders. A strategy consists of a set of intentional elements and indicators that are given initial satisfaction values. These satisfaction values capture contextual or future situations as well as choices among alternative means of reaching various goals. For indicators, these satisfaction values are based on real-world measurements. These satisfaction values are then propagated to the other intentional elements through their links, enabling a global assessment of the strategy being studied as well as the global satisfaction of the actors involved. A good strategy provides rationale and documentation for decisions leading to requirements, providing better context for standards/system developers and implementers while avoiding unnecessary re-evaluations of worse alternative strategies.

GRL also provides support for reasoning about scenarios by establishing correspondences between intentional GRL elements and non-intentional elements referring to scenario models of URN-FR. Modelling both goals and scenarios is complementary and aids the identification of further goals and additional scenarios (and scenario steps) important to stakeholders, thus contributing to the completeness and accuracy of requirements.

## **1.2 Scenario modelling with URN**

The subset of the URN language that addresses ITU-T Z.150 URN-FR language requirements is named Use Case Map (UCM). UCM specifications employ scenario paths to illustrate causal relationships among responsibilities. Furthermore, UCMs provide an integrated view of behaviour and structure by allowing the superimposition of scenario paths on a structure of abstract components. The combination of behaviour and structure enables architectural reasoning after which it is possible to refine UCM specifications into more detailed scenario models such as MSCs or UML sequence diagrams, or into state machines in SDL-2010 or UML statechart diagrams and finally into concrete implementations. Validation, verification, performance analysis, interaction detection and test generation can be performed at all stages. Thus, the UCM notation enables a seamless transition from the informal to the formal by bridging the modelling gap between goal models and natural language requirements (e.g., use cases) and design in an explicit and visual way. The UCM notation allows the modeller to delay the specification of component states and messages and even, if desired, of concrete components to later, more appropriate, stages of the development process. The goal of the UCM notation is to provide the right degree of formality at the right time in the development process.

UCM specifications identify input sources and output sinks, as well as describe the required inputs and outputs of a scenario. UCM specifications also integrate many scenarios or related use cases in a map-like diagram. Scenarios can be structured and integrated incrementally. This enables reasoning about and detection of potential undesirable interactions of scenarios and components. Furthermore, the dynamic (runtime) refinement capabilities of the UCM notation allow for the specification of (runtime) policies and for the specification of loosely coupled systems where functionality is decided at runtime through negotiation between components or compliance to high-level goals. UCM scenarios can be integrated together, yet individual scenarios are tractable through scenario definitions based on a simple data model. UCMs treat scenario paths as first class model entities and

therefore build the foundation to more formally facilitate reusability of scenarios and behavioural patterns across a wide range of architectures.

The UCM notation is a specification language intended for modellers as well as non-specialists because of its visual, simple and intuitive nature but at the same time it aims to provide sufficient rigorousness for developers or tools and contracts.

Most of the characteristics of excellent requirements such as verifiable, complete, consistent, unambiguous, understandable, modifiable and traceable can be supported by UCMs. Others such as prioritized and annotated are easily incorporated.

### 1.3 Documentation structure

This Recommendation defines the User Requirements Notation in the following way:

- Clauses 2, 3 and 4 describe, respectively, references to related ITU-T Recommendations and other standards, definitions and acronyms used in this Recommendation.
- Clause 5 describes conventions used in this Recommendation, with a particular emphasis on metamodelling.
- Clause 6 specifies the abstract syntax of basic structural features of the URN language.
- Clause 7 specifies the abstract syntax, concrete graphical syntax and semantics of GRL features. A concrete textual syntax of GRL is provided in Annex B.
- Clause 8 specifies the abstract syntax, concrete graphical syntax and semantics of UCM features. A concrete textual syntax of UCM is provided in Annex B.
- Clause 9 specifies the data language used to formalize conditions and expressions.
- Clause 10 specifies an XML-based interchange format for URN models based on the concrete syntax metamodel. The XML schema definition is provided in Annex A.
- Clause 11 describes basic URN analysis techniques, namely GRL model evaluation and UCM scenario path traversal.
- Clause 12 presents how this Recommendation complies with [ITU-T Z.150].
- Clause 13 defines levels of compliances for tools.
- Annex A presents the XML schema definition of the URN interchange format.
- Annex B presents the definition of the Textual User Requirements Notation (TURN).
- Appendix I provides a summary of the URN metamodel and graphical notation.
- Appendix II gives three examples of GRL model evaluation algorithms.
- Appendix III gives two examples of UCM path traversal mechanisms.

## 2 References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published. The reference to a document within this Recommendation does not give it, as a stand-alone document, the status of a Recommendation.

[ITU-T T.55] Recommendation ITU-T T.55 (2008), *Use of the universal multiple-octet coded character set (UCS)*.

[ITU-T Z.104] Recommendation ITU-T Z.104 (2011), *Specification and Description Language – Data and action language in SDL-2010*.

- [ITU-T Z.111] Recommendation ITU-T Z.111 (2008), *Notations and guidelines for the definition of ITU-T languages*.
- [ITU-T Z.150] Recommendation ITU-T Z.150 (2011), *User Requirements Notation (URN) – Language requirements and framework*.
- [W3C XSD1] W3C (2004), *XML Schema Part 1: Structures Second Edition*.  
<<http://www.w3.org/TR/xmlschema-1>>
- [W3C XSD2] W3C (2004), *XML Schema Part 2: Datatypes Second Edition*.  
<<http://www.w3.org/TR/xmlschema-2>>

### 3 Definitions

#### 3.1 Terms defined elsewhere

The definitions given in [ITU-T Z.150] apply.

#### 3.2 Terms defined in this Recommendation

This Recommendation defines the following terms:

**3.2.1 abort scope:** (UCM) the set of maps comprising the map where the abort start point is defined as well as all lower level maps as established by the hierarchy of stubs and plug-in maps.

**3.2.2 actor:** (GRL) element that represents an active entity (stakeholder or other) that has intentions and carries out actions to achieve its goals by exercising its know-how.

**3.2.3 Goal-oriented Requirement Language (GRL):** The subset of the User Requirements Notation used to model and analyse non-functional requirements and quality attributes with goal graphs.

**3.2.4 GRL link:** (GRL) intentional relationship existing between intentional elements or actors.

**3.2.5 indicator:** (GRL) relates a measured, real-world, quantitative or qualitative value with a quantitative or qualitative satisfaction value and is used to measure the performance, any other relevant real-world property or satisfaction of an intentional element, actor or system.

**3.2.6 in-path:** (UCM) incoming path of a stub, in particular the last node connection before reaching the stub.

**3.2.7 intentional element:** (GRL) element that describes an intention. Used for models that allow answering questions such as why particular behaviours, informational and structural aspects were chosen to be included in the system requirement, what alternatives were considered, what criteria were used to deliberate among alternative options and what the reasons were for choosing one alternative over the other.

**3.2.8 out-path:** (UCM) outgoing path of a stub, in particular the first node connection after leaving the stub.

**3.2.9 release path:** (UCM) path with an end point or empty point that is connected to a timer.

**3.2.10 scenario definition:** (UCM) collection of initial values, initial conditions and desired conditions used to identify and test individual scenarios during the traversal of a UCM model.

**3.2.11 strategy:** (GRL) collection of satisfaction values associated with intentional elements used to provide an initial context for GRL model analysis.

**3.2.12 traversal root map:** (UCM) map that is at the highest level in the map hierarchy established by the traversal mechanism for a scenario definition.

**3.2.13 trigger path:** (UCM) path with an end point or empty point that is connected to a waiting place.

**3.2.14 unconnected start point:** (UCM) start point that is not a failure start point or abort start point and that is not directly connected to another end point or to another path.

**3.2.15 Use Case Map notation:** The subset of the User Requirements Notation used to model and analyse operational requirements and functional requirements with use cases and scenarios.

**3.2.16 visit:** (UCM) visit of a synchronizing stub is characterized by how often an in-path of the stub has been traversed. If an in-path is traversed the first time, then it is the first visit of the stub. If the same in-path is traversed the  $n^{\text{th}}$  time, then it is the  $n^{\text{th}}$  visit of the stub. If another in-path of the stub is traversed for the first time, then it is the first visit of the stub. Plug-in maps that have been instantiated because of a visit are said to belong to the visit.

**3.2.17 waiting path:** (UCM) incoming path of a waiting place or timer that is not a release path or trigger path.

#### 4 Abbreviations and acronyms

This Recommendation uses the following abbreviations:

COTS	Commercial-Off-The-Shelf
DSP	Digital Signal Processor
FAS	Failure start point or Abort Start point
FP	Failure Point
FR	Functional Requirement
GRL	Goal-oriented Requirement Language
MSC	Message Sequence Chart
NFR	Non-Functional Requirement
NTRP	Number of arrivals along the Trigger/Release Path
NWP	Number of arrivals along the Waiting Path
OCS	Originating Call Screening
PCC	Path Continuation Criteria
PT	Path Traversal
RP	Regular Path
SDL-2010	Specification and Description Language 2010
TRP	Trigger/Release Path
TTCN-3	Testing and Test Control Notation (TTCN-3)
TURN	Textual User Requirements Notation
UCM	Use Case Map
UCS	Universal Multiple-Octet Coded Character Set
UML	Unified Modelling Language
URN	User Requirements Notation
URN-FR	User Requirements Notation – Functional Requirements
URN-NFR	User Requirements Notation – Non-Functional Requirements
UTF-8	8-bit UCS/Unicode Transformation Format

WP	Waiting Path
XML	eXtensible Markup Language
XSD	XML Schema Definition

## 5 Conventions

The conventions of [ITU-T Z.111] apply to this Recommendation.

This Recommendation uses the universal multiple-octet coded character set (UCS) encoding of characters recommended in [ITU-T T.55].

### 5.1 Grammars

The conventions of [ITU-T Z.111] apply to this Recommendation.

### 5.2 Basic definitions

#### 5.2.1 Validity

A specification is a valid User Requirements Notation specification only if it satisfies the syntactic rules and the static conditions defined in this Recommendation.

### 5.3 Presentation style

The conventions of [ITU-T Z.111] apply to this Recommendation.

#### 5.3.1 Division of text

The conventions of [ITU-T Z.111] apply to this Recommendation.

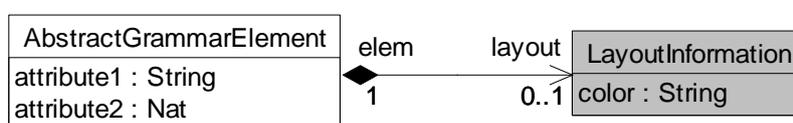
#### 5.3.2 Titled enumeration items

##### a) *Abstract grammar*

The abstract grammar is specified in the form defined in [ITU-T Z.111]. The metamodel presentation of abstract syntax in [ITU-T Z.111], clause 5.4.1.2 is used. Attributes, relationships to other metaclasses and constraints (static conditions expressed in natural language) are specified for each metaclass in the metamodel. As an extension to [ITU-T Z.111] conventions there are additional subheadings of the form i) **Attributes**, ii) **Relationships** and iii) **Constraints**.

##### b) *Concrete grammar*

The URN concrete grammar is presented as an extension to the abstract grammar metamodel combined with a description of the graphical symbols used. The concrete grammar includes all the metaclasses (with attributes, relationships and constraints) of the abstract grammar. The additional concepts (shown as grey metaclasses) that extend the abstract grammar metamodel are useful to support a graphical language but they have no semantic implication. Common additional concepts include layout information, line styles and informal descriptions. For example, in Figure 1, a colour attribute is added to an element of the abstract grammar. Composition with multiplicity 0..1 is used here to ensure that specifications without this layout information are still valid and that the additional concept will not interfere during analysis.



**Figure 1 – Example of metaclasses from an abstract grammar (white) and a concrete grammar (grey)**

As a convention and in order to simplify diagrams, metaclasses that are defined in a different section but that are referenced in a diagram will have their attributes hidden. For example, Figure 2 makes reference to an abstract grammar element metaclass defined elsewhere (e.g., in Figure 1), but its attributes are hidden. Note that this is different from a metaclass *without* attributes, in which case the attribute compartment at the bottom of the metaclass (bounded by a horizontal line in the middle of the rectangle) is empty. Similarly, only relevant associations are presented in diagrams. Appendix I provides a global overview of all metaclasses, their associations and their attributes.



**Figure 2 – Example of metaclasses where the left one (with attributes hidden) is detailed elsewhere**

Not all URN metaclasses, attributes or relationships have a concrete graphical notation. It is then up to tools to provide ways of creating, accessing and modifying instances of these metaclasses (for instance, through a "property" window).

Many elements with a graphical representation also have model-specific coordinates and sizes. The following convention is used for layout coordinates information.

- Horizontal coordinate (x-axis): An integer value representing the number of point units from the origin (0). Positive values are at the right of the origin and negative values at the left of the origin.
- Vertical coordinate (y-axis): An integer value representing the number of point units from the origin (0). Positive values are below the origin and negative values above the origin.

*c) Semantics*

The semantics of the abstract grammar metaclass is expressed in natural language. The semantics of a concrete grammar metaclass is that of its abstract grammar metamodel elements (the additional grey metaclasses have no semantics).

*d) Model*

This clause, when present, describes shorthand or alternative concrete syntaxes.

*e) Example*

Where necessary, examples of use are included. These examples are only informative, not normative.

## 6 URN basic structural features

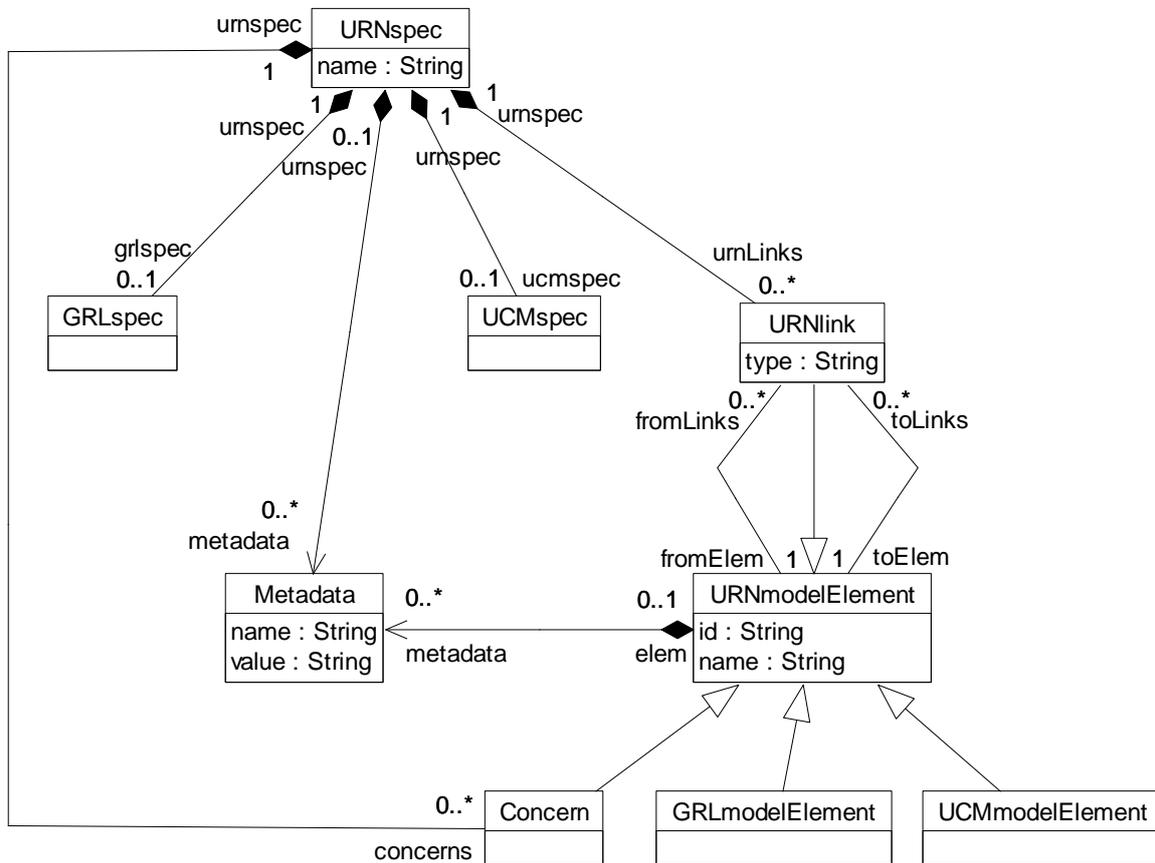
The URN basic structural features describe containers for URN, GRL and UCM specifications, as well as definitions of URN model elements, their links and metadata, concerns and conditions. The abstract syntax metaclasses are first presented in clause 6.1. Their concrete grammar references concrete syntax metaclasses regrouped in clause 6.2.

- Clause 6.1: URN abstract grammar metaclasses
- Clause 6.2: URN concrete grammar metaclasses

NOTE – Clause 9 defines the data model and data language for URN. In particular, it defines the *Integer* type used in some of the attributes of the abstract and concrete syntax metaclasses for URN. The attribute types Boolean, Nat and String used in these metaclasses are those defined in [ITU-T Z.111].

## 6.1 URN abstract grammar metaclasses

The topmost metaclass, *URNspec* (see Figure 3), contains directly or indirectly all the other elements of a URN model, including concerns (see Figure 4). In this Recommendation, the terms "URN model" and "URN specification" are used interchangeably.



**Figure 3 – Abstract grammar: URN specification, links, metadata and model elements**

### 6.1.1 URNspec

*URNspec* is the root element of a URN model/specification. It names the specification and serves as a container for all the other specification elements (see Figure 3).

#### a) Abstract grammar

##### i) Attributes

- name (String): The name of the URN specification.

##### ii) Relationships

- Composition of *GRLspec* (0..1): The *URNspec* may contain one GRL specification (see clause 7.1.1).
- Composition of *UCMspec* (0..1): The *URNspec* may contain one UCM specification (see clause 8.1.1).
- Composition of *URNlink* (0..\*): The *URNspec* may contain URN links.
- Composition of *Metadata* (0..\*): The *URNspec* may contain metadata information.
- Composition of *Concern* (0..\*): The *URNspec* may contain concerns.

### iii) Constraints

- There exists exactly one instance of *URNspec* in a URN specification.

### b) Concrete grammar

*URNspec* does not have a graphical visual representation. However, it may contain additional information in an instance of *ConcreteURNspec*. Optionally, the name of *URNspec* is shown in the top left corner of a *GRLGraph* or *UCMmap* before the additional information (see *ConcreteURNspec*).

### i) Relationships

- Composition of *ConcreteURNspec* (0..1): The *URNspec* may contain one concrete URN specification (see Figure 5).

### c) Semantics

None (*URNspec* is a structural concept only).

## 6.1.2 URNmodelElement

URN model elements have names and unique identifiers. They can also be linked to each other (see Figure 3).

### a) Abstract grammar

#### i) Attributes

- *id* (String): The identifier of the URN model element.
- *name* (String): The name of the URN model element.

#### ii) Relationships

- Composition of *Metadata* (0..\*): A *URNmodelElement* may contain metadata information.
- Association with *URNlink* (*fromLinks*, 0..\*): A *URNmodelElement* may be the source of URN links.
- Association with *URNlink* (*toLinks*, 0..\*): A *URNmodelElement* may be the target of URN links.
- Association with *Concern* (0..1): A *URNmodelElement* may belong to one concern.
- *URNmodelElement* is a superclass of *URNlink*, *Concern*, *GRLmodelElement* (see clause 7.1.2) and *UCMmodelElement* (see clause 8.1.2).

#### iii) Constraints

- *id* shall be unique within the URN specification.
- All instances of *URNmodelElement* shall appear in one of its subclasses (that is, metaclass *URNmodelElement* is abstract).

### b) Concrete grammar

The *id* of a *URNmodelElement* does not have a concrete syntax and is automatically assigned to ensure its uniqueness. The concrete syntax for *URNmodelElement* is further defined in its subclasses. In addition, a *URNmodelElement* may contain an informal description in a *Description*.

#### i) Relationships

- Composition of *Description* (0..1): A *URNmodelElement* may contain one description (see Figure 6).

c) *Semantics*

A *URNmodelElement* is a uniquely identifiable model element that can contain metadata and be linked to other model elements. Its subclasses possibly have additional attributes and relationships.

### 6.1.3 URNlink

A URN link is a URN model element that connects a source URN model element to a target URN model element. URN links have a user-defined type (see Figure 3).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *URNmodelElement*.
- type (String): The user-defined type of the URN link.

**ii) Relationships**

- Inherits relationships from *URNmodelElement*.
- Contained by *URNspec* (1): A *URNlink* is contained in the URN specification.
- Association with *URNmodelElement* (fromElem, 1): A *URNlink* has one source URN model element.
- Association with *URNmodelElement* (toElem, 1): A *URNlink* has one target URN model element.

**iii) Constraints**

- Inherits constraints from *URNmodelElement*.

b) *Concrete grammar*

The presence of a link on a source or target model element is indicated with a triangle symbol (►) next to the name of the element, if that element's name is displayed in the concrete syntax. For a source model element, the triangle points to the right, whereas for a target model element the triangle points to the left. Optionally, a *URNlink* is visualized using the textual syntax for a *URNlink* as specified in Annex B.

**i) Relationships**

- Inherits relationships from *URNmodelElement*.

c) *Semantics*

*URNlinks* provide modellers with a way to create new relationships of various types between any pair of model elements in a URN specification. These links can be used for traceability, refinement, composition and other purposes, hence providing an extensible semantics to URN.

### 6.1.4 Metadata

*Metadata* is a name-value pair that can be used to attach information to a URN specification or its model elements. *Metadata* is contained by the *URNspec* or a *URNmodelElement* (see Figure 3).

a) *Abstract grammar*

**i) Attributes**

- name (String): The name of the URN metadata information instance.
- value (String): The value of the URN metadata information instance.

**ii) Relationships**

- None.

### iii) Constraints

- Each *Metadata* instance is contained in exactly one instance of type *URNspec* or *URNmodelElement*.

### b) Concrete grammar

*Metadata* does not have a graphical visual representation. Optionally, a *Metadata* is visualized using the textual syntax for a *Metadata* as specified in Annex B.

### c) Semantics

*Metadata* instances provide modellers with a way to attach user-defined named values to most elements found in a URN specification, hence providing an extensible semantics to URN.

## 6.1.5 Concern

A *Concern* is a guarded grouping of URN model elements. Concerns are typically used to group related GRL and UCM diagrams into one unit of understanding (see Figures 3 and 4).

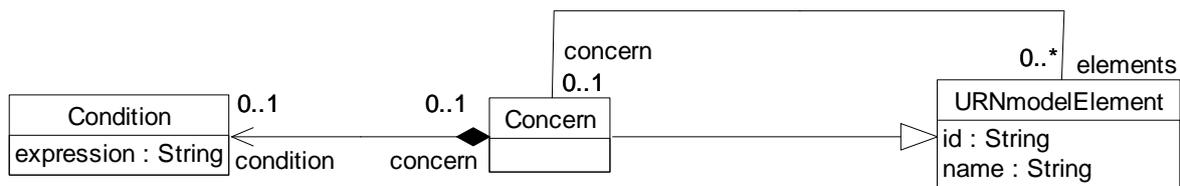


Figure 4 – Abstract grammar: URN concerns

### a) Abstract grammar

#### i) Attributes

- Inherits attributes from *URNmodelElement*.

#### ii) Relationships

- Inherits relationships from *URNmodelElement*.
- Contained by *URNspec* (1): A *Concern* is contained in the URN specification.
- Composition of *Condition* (0..1): A *Concern* may contain one condition.
- Association with *URNmodelElement* (0..\*): A *Concern* possibly groups URN model elements.

#### iii) Constraints

- Inherits constraints from *URNmodelElement*.

### b) Concrete grammar

A *Concern* does not have a graphical visual representation. Optionally, a *Concern* is visualized using the textual syntax for a *Concern* as specified in Annex B.

#### i) Relationships

- Inherits relationships from *URNmodelElement*.

### c) Semantics

A *Concern* groups URN model elements together. This grouping can be guarded with a *Condition* for composition purposes.

## 6.1.6 Condition

*Condition* is a Boolean expression that serves as a guard, precondition, postcondition or failure condition. A *Condition* is contained by one of the following model elements: *Concern*, *StartPoint*, *EndPoint*, *PluginBinding*, *NodeConnection* or *ScenarioDef* (see Figures 4, 77 and 92).

### a) Abstract grammar

#### i) Attributes

- expression (String): The Boolean expression of the condition.

#### ii) Relationships

- None.

#### iii) Constraints

- The expression of a *Condition* shall be a Boolean expression, as defined in clause 9.3.
- Each *Condition* instance is contained in exactly one instance of type *Concern*, *StartPoint*, *EndPoint*, *PluginBinding*, *NodeConnection* or *ScenarioDef*.

### b) Concrete grammar

*Condition* has no concrete syntax, but the label of the *ConcreteCondition* contained by the condition is visualized for *NodeConnections*, *StartPoints* and *EndPoints* (see clauses 8.2.3, 8.2.6 and 8.2.8, respectively). Conditions for *Concerns*, *PluginBindings* or *ScenarioDefs* are optionally visualized using the textual syntax for a *Concern*, *PluginBinding*, or *ScenarioDef*, respectively, as specified in Annex B.

#### i) Relationships

- Composition of *ConcreteCondition* (0..1): A *Condition* may have one concrete condition (see Figure 7).
- Composition of *Label* (0..1): A *Condition* may have one label (see Figure 52).

### c) Semantics

The expression of a *Condition* contained by a *Concern* indicates whether the grouping of model elements identified by the concern is to be enabled in the URN specification (true) or disabled (false).

The expression of a *Condition* not contained by a *Concern* is evaluated at runtime when the model element to which the condition belongs is reached during the path traversal of the UCM specification. The evaluation results in either true or false.

The expression may make use of globally defined *Variables* and shall be well-formed according to the textual grammar detailed in clause 9.3. If the expression uses a variable name with a "\_pre" suffix (e.g., in the postcondition of a scenario definition), then the initialized value of this variable, prior to traversal of the UCM specification, is used. This is mainly useful in postconditions that compare the runtime value of a variable with the initial value of that variable. For example, the expression `VariableX == VariableX_pre + 1` will be true if `VariableX` has been incremented by 1 since the beginning of the traversal.

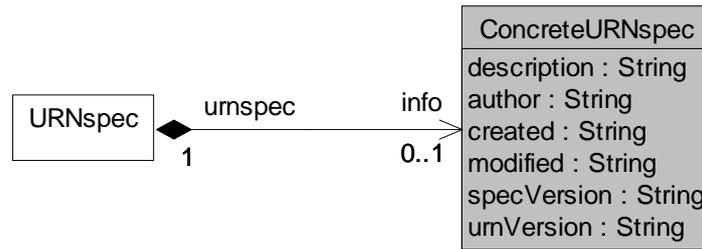
If a *Condition* is not specified for a *Concern*, *StartPoint*, *EndPoint*, *PluginBinding*, *NodeConnection* or *ScenarioDef*, but the existence of the condition is required by, for example, model analysis, then the evaluation of the required condition is assumed to result in true.

## 6.2 URN concrete grammar metaclasses

The following concrete grammar metaclasses may be contained by some of the abstract grammar metaclasses. They have no semantics.

### 6.2.1 ConcreteURNspec

The *ConcreteURNspec* metaclass contains standard meta-information about the URN model (*URNspec*) itself (see Figure 5).



**Figure 5 – Concrete grammar: ConcreteURNspec metaclass**

a) *Abstract grammar*

None. This is a concrete syntax metaclass only.

b) *Concrete grammar*

There is no graphical visual representation of this metaclass. Optionally, the meta-information is visualized using the textual syntax for *ConcreteURNspec* as specified in Annex B and shown after the name of the *URNspec* in the top left corner of a *GRLGraph* or *UCMmap*.

**i) Attributes**

- description (String): An informal description of the URN specification.
- author (String): The author of the URN specification.
- created (String): The date and time of creation of the URN specification. The suggested format is (in English) "Month day, year hours:minutes:seconds AmOrPm timezone". For example: "November 15, 2007 9:21:06 AM EST".
- modified (String): The date and time of the last modification to the URN specification. The suggested format is (in English) "Month day, year hours:minutes:seconds AmOrPm timezone". For example: "November 15, 2007 9:21:06 AM EST".
- specVersion (String): The version number of the URN specification. It is suggested to use an integer that starts at 1 when the specification is first created and that is incremented by one each time the specification is modified.
- urnVersion (String): The version number of the URN standard used. For example: "Z.151 (11/08)".

**ii) Constraints**

- The date modified is later than the date created.

### 6.2.2 Description

An informal *Description* can be attached to any *URNmodelElement* (see Figure 6).



**Figure 6 – Concrete grammar: Description metaclass**

a) *Abstract grammar*

None. This is a concrete syntax metaclass only.

b) *Concrete grammar*

There is no graphical visual representation of this metaclass. Optionally, the description of a *URNmodelElement* preceded by "description:" is visualized next to the *URNmodelElement*.

i) *Attributes*

- description (String): An informal description of the URN model element.

### 6.2.3 ConcreteCondition

*ConcreteCondition* defines a label and a description for a *Condition* (see Figure 7).



**Figure 7 – Concrete grammar: ConcreteCondition metaclass**

a) *Abstract grammar*

None. This is a concrete syntax metaclass only.

b) *Concrete grammar*

The label of *ConcreteCondition* is visualized for containing *Conditions* of *NodeConnections*, *StartPoints* and *EndPoints* (see clauses 8.2.3, 8.2.6 and 8.2.8, respectively). Optionally, the label of *ConcreteCondition* is visualized using the textual syntax for containing *Conditions* of a *Concern*, *PluginBinding*, or *ScenarioDef* as specified in Annex B.

i) **Attributes**

- label (String): The label for the condition is used for visualization purposes.
- description (String): An informal description of the condition.

c) *Semantics*

None.

## 7 GRL features

The Goal-oriented Requirement Language provides a set of URN features that enable the description and analysis of goals/intentions of systems and stakeholders. The GRL features are grouped under six categories:

- GRL basic structural features: clause 7.1.
- GRL actors: clause 7.2.
- GRL intentional elements: clause 7.3.
- GRL links: clause 7.4.
- GRL strategies: clause 7.5.
- GRL indicators: clause 7.6.
- GRL contribution contexts: clause 7.7.
- GRL concrete grammar metaclasses: clause 7.8.

NOTE – Many of the concrete grammar metaclasses defined here are also used by UCM features.

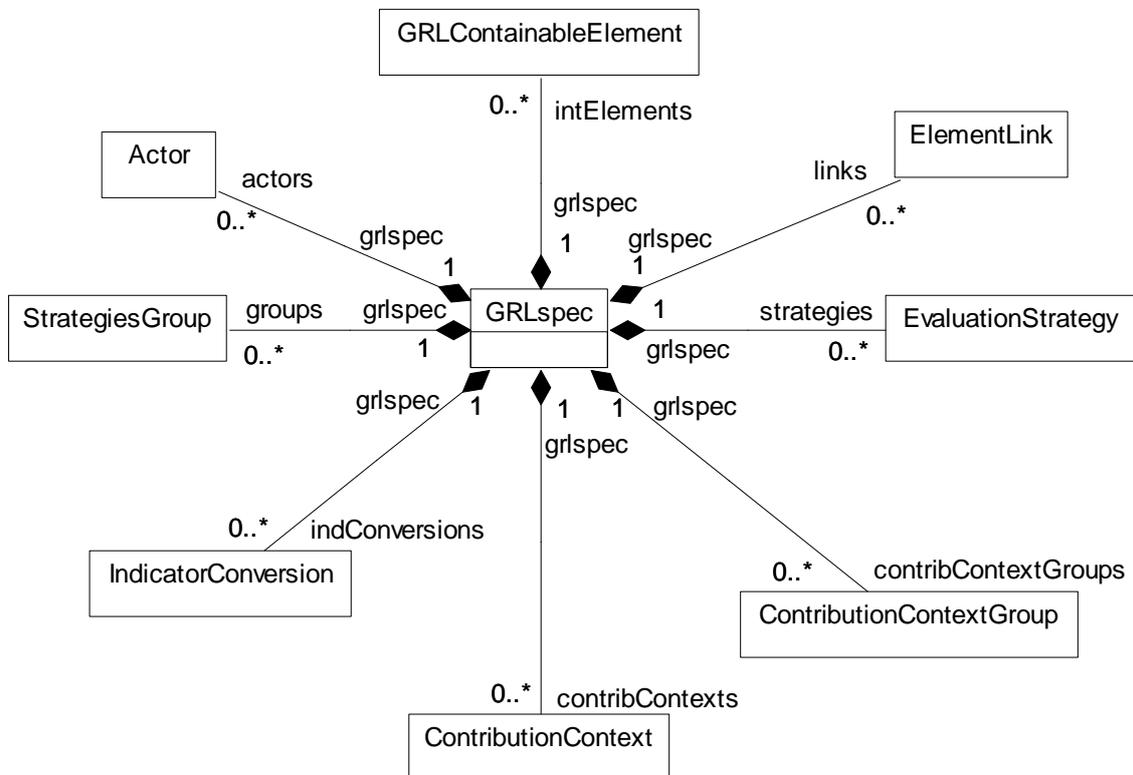
## 7.1 GRL basic structural features

The GRL basic structural features describe containers for GRL specifications, as well as definitions of GRL model elements, including linkable elements. The abstract grammar metaclasses are presented in this clause, whereas their concrete grammar metaclasses are detailed in clause 7.8.

### 7.1.1 GRLspec

*GRLspec* serves as a container for the GRL specification elements (see Figure 8).

NOTE – The name *intElements* is used for backward compatibility with previous versions of this Recommendation. A more appropriate name would be *contElements*.



**Figure 8 – Abstract grammar: GRL specification**

#### a) Abstract grammar

##### i) Attributes

- None.

##### ii) Relationships

- Contained by *URNspec* (1): The *GRLspec* is contained in the URN specification (see Figure 3).
- Composition of *Actor* (0..\*): The *GRLspec* may contain actor definitions.
- Composition of *GRLContainableElement* (0..\*): The *GRLspec* may contain containable elements.
- Composition of *IndicatorConversion* (0..\*): The *GRLspec* may contain conversions for indicators.
- Composition of *ElementLink* (0..\*): The *GRLspec* may contain element links.
- Composition of *StrategiesGroup* (0..\*): The *GRLspec* may contain strategy groups.

- Composition of *EvaluationStrategy* (0..\*): The *GRLspec* may contain evaluation strategies.
- Composition of *ContributionContextGroup* (0..\*): The *GRLspec* may contain contribution context groups.
- Composition of *ContributionContext* (0..\*): The *GRLspec* may contain contribution contexts.

### iii) Constraints

- None.

### b) Concrete grammar

*GRLspec* has no concrete syntax. However, it may contain concrete GRL specification information and GRL graphs.

#### i) Relationships

- Composition of *ConcreteGRLspec* (0..1): The *GRLspec* may contain one concrete GRL specification (see Figure 30).
- Composition of *GRLGraph* (0..\*): The *GRLspec* may contain GRL graphs (see Figure 30).

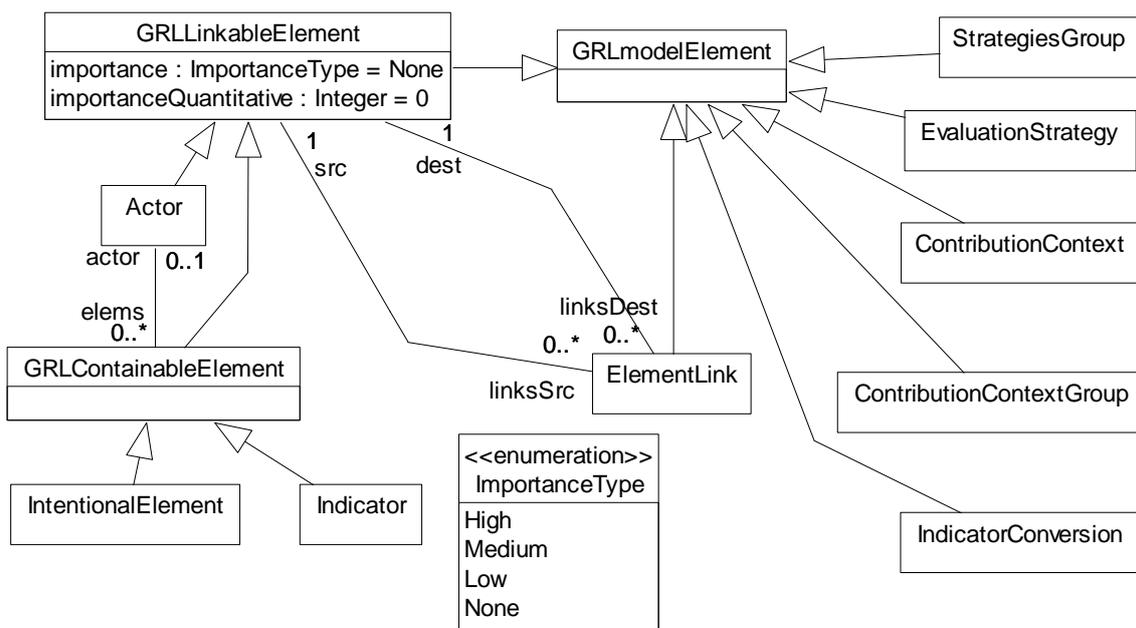
### c) Semantics

None (*GRLspec* is a structural concept only).

## 7.1.2 GRLmodelElement

### a) Abstract grammar

A *GRLmodelElement* is a URN model element specialized for GRL concepts (see Figure 9).



**Figure 9 – Abstract grammar: GRL model, linkable and containable elements**

#### i) Attributes

- Inherits attributes from *URNmodelElement* (see Figure 3).

## ii) Relationships

- Inherits relationships from *URNmodelElement*.
- *GRLmodelElement* is a superclass of *GRLLinkableElement*, *ElementLink*, *StrategiesGroup*, *EvaluationStrategy*, *IndicatorConversion*, *ContributionContextGroup* and *ContributionContext*.

## iii) Constraints

- Inherits constraints from *URNmodelElement*.
- All instances of *GRLmodelElement* shall appear in one of its subclasses (that is, metaclass *GRLmodelElement* is abstract).

### b) Concrete grammar

The concrete syntax for *GRLmodelElement* is further defined in its subclasses.

## i) Relationships

- Inherits relationships from *URNmodelElement* (see Figure 6).
- *GRLmodelElement* is a superclass of *GRLGraph*, *ActorRef*, *GRLNode* and *LinkRef* (see Figure 30).

### c) Semantics

A *GRLmodelElement* is a uniquely identifiable GRL model element that can contain metadata and be linked to other URN model elements through *URNlinks*. Its subclasses may have additional attributes and relationships.

## 7.1.3 GRLLinkableElement

A *GRLLinkableElement* is a GRL model element that has an importance and can be linked to other GRL linkable elements through an *ElementLink*. *GRLLinkableElement* abstracts the commonalities of actor definitions, intentional elements and indicators (see Figure 9).

### a) Abstract grammar

## i) Attributes

- Inherits attributes from *GRLmodelElement*.
- *importance* (*ImportanceType*): Qualitative importance of the containable element to its containing actor definition in the case of intentional elements and indicators and to the overall GRL model in the case of actors, if any. Default value is *None*.
- *importanceQuantitative* (*Integer*): Quantitative importance of the containable element to its containing actor definition in the case of intentional elements and indicators and to the overall GRL model in the case of actors, if any. Default value is 0 (see clause 9.2.2).

## ii) Relationships

- Inherits relationships from *GRLmodelElement*.
- Association with *ElementLink* (*linksSrc*, 0..\*): A *GRLLinkableElement* may be the source of GRL element links.
- Association with *ElementLink* (*linksDest*, 0..\*): A *GRLLinkableElement* may be the destination of GRL element links.
- *GRLLinkableElement* is a superclass of *GRLContainableElement* and *Actor*.
- Uses *ImportanceType* enumeration.

## iii) Constraints

- Inherits constraints from *GRLmodelElement*.

- All instances of *GRLLinkableElement* shall appear in one of its subclasses (that is, metaclass *GRLLinkableElement* is abstract).
- $\text{importanceQuantitative} \geq 0$  and  $\text{importanceQuantitative} \leq 100$ .
- The name of a *GRLLinkableElement* cannot be the empty String "".

b) *Concrete grammar*

The concrete syntax for *GRLLinkableElement* is further defined in its subclasses. The line and fill colours of a linkable element are specified in its concrete style (*ConcreteStyle*) and are hence shared by all the linkable element's references.

i) **Relationships**

- Inherits relationships from *GRLmodelElement*.
- Composition of *ConcreteStyle* (0..1): A *GRLLinkableElement* may contain one concrete style (see Figure 30).

c) *Semantics*

A *GRLLinkableElement* is a GRL model element that can be linked to other actor definitions, intentional elements and indicators. The semantics of importance and importanceQualitative are described by the subclasses of *GRLLinkableElement* (see clauses 7.1.5 and 7.2.1).

#### 7.1.4 ImportanceType

The qualitative importance of an actor to the overall GRL model and of an intentional element or indicator to its actor definition can be High, Medium, Low or None (see Figure 9).

a) *Abstract grammar*

i) **Attributes**

- None (enumeration metaclass).

ii) **Relationships**

- Used by *GRLLinkableElement*.

iii) **Constraints**

- None.

b) *Concrete grammar*

None (enumeration metaclass). However, it influences the presentation of intentional elements and indicators contained by actor definitions (see clause 7.8.5) as well as the presentation of actors (see clauses 7.8.3 and 7.8.6).

c) *Semantics*

High is used for specifying the highest importance, Low for some non-null importance, Medium for a level in between high and low, and finally None for no importance. The satisfaction level of an intentional element or indicator with a None importance will have no impact on the qualitative evaluation of the global satisfaction of the associated actor definition. The satisfaction level of an actor with a None importance will have no impact on the qualitative evaluation of the global satisfaction of the overall GRL model.

#### 7.1.5 GRLContainableElement

A *GRLContainableElement* is a GRL model element that can be contained in an actor. *GRLContainableElement* abstracts the commonalities of intentional elements and indicators (see Figure 9). A *GRLContainableElement* may be given *Evaluation* values (see Figure 24).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *GRLLinkableElement*.

**ii) Relationships**

- Inherits relationships from *GRLLinkableElement*.
- Association with *Actor* (0..1): A *GRLContainableElement* may be contained in one actor definition.
- *GRLContainableElement* is a superclass of *IntentionalElement* and *Indicator*.

**iii) Constraints**

- Inherits constraints from *GRLLinkableElement*.
- All instances of *GRLContainableElement* shall appear in one of its subclasses (that is, metaclass *GRLContainableElement* is abstract).
- If importance of a *GRLContainableElement* is greater than 0, then the *GRLContainableElement* shall be contained in an actor definition.
- If importanceQuantitative is not equal to None, then the *GRLContainableElement* shall be contained in an actor definition.

b) *Concrete grammar*

The concrete syntax for *GRLContainableElement* is further defined in its subclasses.

**i) Relationships**

- Inherits relationships from *GRLLinkableElement*.
- Association with *IntentionalElementRef* (0..\*): An *GRLContainableElement* may be referenced by intentional (and indicator) element references (see Figure 30).

c) *Semantics*

A *GRLContainableElement* is a GRL model element that can be contained in an actor definition.

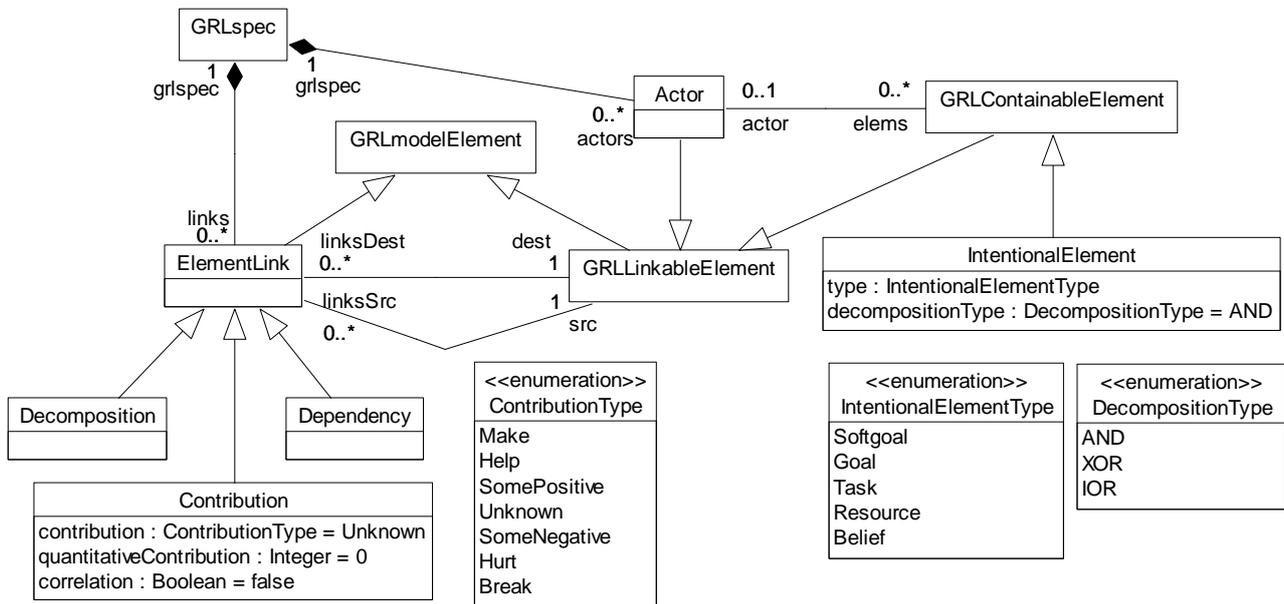
A value of 0 for importanceQuantitative means that the intentional element or indicator is not important to the actor definition, whereas a value of 100 means that it is highly important. A value of None for importance means that the intentional element or indicator is not important to the actor definition, whereas a value of High means that it is highly important. Often, only top-level intentional elements or indicators in a GRL actor will have a non-null importance factor, which summarizes the importance of decomposing or contributing elements.

The two importance attributes are only taken into consideration during actor satisfaction analysis, when the intentional element or indicator is included by an actor definition (they have no meaning otherwise). Only the relevant importance attribute is considered depending on the type of analysis (qualitative or quantitative).

It is not required for importance and importanceQuantitative to be consistent, as modellers may want to use only one type of analysis (qualitative or quantitative). However, it is recommended to keep them consistent if the modellers intend to switch between different types of analysis.

## 7.2 GRL actors

Figure 10 shows the metaclasses for GRL actors, intentional elements and their links. It is referenced by this clause as well as by clauses 7.3 and 7.4.



**Figure 10 – Abstract grammar: GRL actors, intentional elements and links**

### 7.2.1 Actor

An *Actor* (also referred to as actor definition) is a GRL linkable element that represents an entity that has intentions and carries out actions to achieve its goals by exercising its know-how. Actor definitions are often used to represent stakeholders as well as systems. Actor definitions may contain containable elements (see Figure 10).

One could start modelling the domain using only actors and dependencies between actors, without containable elements inside the actors. One can then add intentional elements and indicators inside the actors to specify why actors depend on each other and how dependencies between actors are fulfilled.

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *GRLLinkableElement*.

**ii) Relationships**

- Inherits relationships from *GRLLinkableElement*.
- Contained by *GRLspec* (1): An *Actor* definition is contained in the GRL specification (see Figure 8).
- Association with *GRLContainableElement* (0..\*): An *Actor* definition may contain intentional elements and indicators.

**iii) Constraints**

- Inherits constraints from *GRLLinkableElement*.
- Any two *Actor* definitions cannot share the same name inside a URN specification.

b) *Concrete grammar*

An actor definition does not have a visual representation, but actor references (*ActorRef*) and collapsed actor references (*CollapsedActorRef*) in GRL diagrams do have a graphical representation. The colour of an actor definition's circle line and the fill colour are defined in the concrete style (*ConcreteStyle*) (from superclass *GRLLinkableElement*) and are hence shared by all the actor's references.

### i) Relationships

- Inherits relationships from *GRLLinkableElement*.
- Association with *ActorRef* (0..\*): An *Actor* definition may be referenced by actor references (see Figure 30).
- Association with *CollapsedActorRef* (0..\*): An *Actor* definition may be referenced by collapsed actor references (see Figure 30).

### c) Semantics

An *Actor* definition is a GRL linkable element that may contain intentional elements and indicators describing its intentions, capabilities and related measures. An actor definition may also depend on another actor definition to satisfy some intentional element or indicator. How well an actor definition is satisfied depends on the satisfaction level and importance of the intentional elements and indicators it contains.

A value of 0 for *importanceQuantitative* means that the actor is not important to the overall GRL model, whereas a value of 100 means that the actor is highly important. A value of *None* for *importance* means that the actor is not important to the overall GRL model, whereas a value of *High* means that the actor is highly important.

The two importance attributes are taken into consideration during overall GRL model satisfaction analysis. Only the relevant importance attribute is considered depending on the type of analysis (qualitative or quantitative).

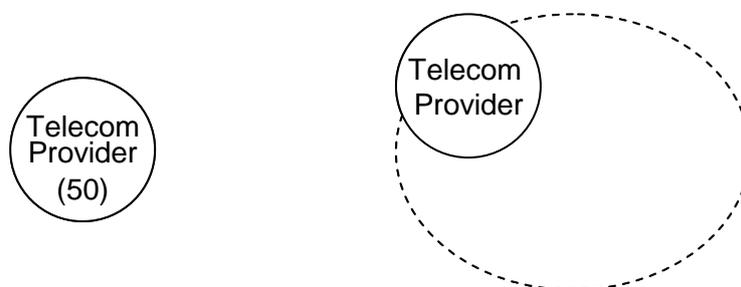
It is not required for *importance* and *importanceQuantitative* to be consistent, as modellers may want to use only one type of analysis (qualitative or quantitative). However, it is recommended to keep them consistent if the modellers intend to switch between different types of analysis.

### d) Model

None.

### e) Examples

Figure 11 is a GRL diagram that shows a "Telecom Provider" as a collapsed actor (left) and as an actor with boundary (right). The collapsed actor has an importance value of 50 whereas the actor with boundary has an importance value of 0 (hence its importance is not shown). See clauses 7.8.3 and 7.8.6 for the details of the concrete syntax.



**Figure 11 – Example: GRL collapsed actor (left) and actor with boundary (right)**

## 7.3 GRL intentional elements

### 7.3.1 IntentionalElement

An *IntentionalElement* is a GRL containable element used for models that allow answering questions such as why particular behaviours, informational and structural aspects were chosen to be included in the system requirement, what alternatives were considered, what criteria were used to deliberate among alternative options and what the reasons were for choosing one alternative over another.

Intentional elements may be included in actor definitions and they can be linked to each other in different ways. There are different types of intentional elements specified. Intentional elements can be decomposed and they can be given a quantitative or qualitative importance level when included in an actor definition (see Figure 10).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *GRLContainableElement*.
- type (*IntentionalElementType*): The type of intentional element.
- decompositionType (*DecompositionType*): The type of decomposition when this intentional element is the source of decomposition link, if any. Default value is AND.

**ii) Relationships**

- Inherits relationships from *GRLContainableElement*.
- Uses *IntentionalElementType* enumeration.
- Uses *DecompositionType* enumeration.

**iii) Constraints**

- Inherits constraints from *GRLContainableElement*.
- If an *IntentionalElement* is associated with an *Actor* definition, then there is only one *IntentionalElement* with this name associated with the *Actor* definition.
- If an *IntentionalElement* is not associated with any *Actor* definition, then there is only one *IntentionalElement* with this name that is not associated with any *Actor* definition.

b) *Concrete grammar*

An intentional element does not have a visual representation, but intentional element references (*IntentionalElementRef*) in GRL diagrams do have a graphical representation.

**i) Relationships**

- Inherits relationships from *GRLContainableElement*.

c) *Semantics*

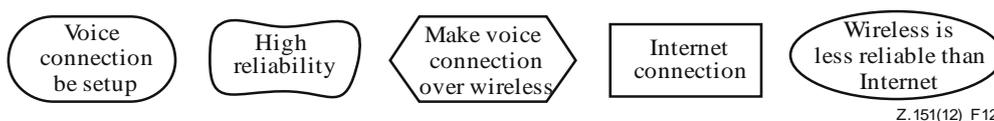
An *IntentionalElement* describes an intention or a capability. An intentional element contained in an *Actor* definition is held by this actor definition and therefore describes part of its intentions or capabilities.

d) *Model*

None.

e) *Examples*

Figure 12 is a GRL diagram that shows five intentional elements, one for each type.



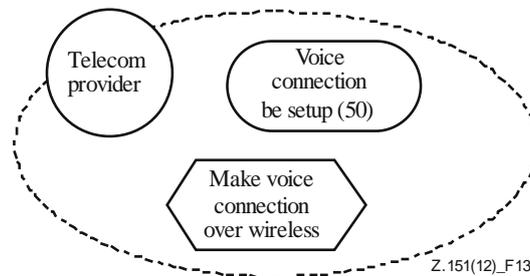
**Figure 12 – Example: GRL intentional elements**

- "Voice Connection Be Setup" is defined as a (hard) goal because this is something that can be achieved entirely.
- "High Reliability" is defined as a softgoal because this is something that can never be entirely achieved (but that can be sufficiently achieved).

- "Make Voice Connection Over Wireless" is defined as a task because this is a particular way of setting up a connection.
- "Internet Connection" is defined as a resource because this is a physical entity that can be available or not.
- "Wireless is less reliable than Internet" is defined as a belief because this provides a rationale for some of the design decisions.

See clauses 7.1.4, 7.3.2, and 7.8.5 for the details of the concrete syntax.

The GRL diagram in Figure 13 shows a goal and a task contained by a "Telecom Provider" actor. The goal has an importance value of 50 whereas the task has an importance value of 0 (hence its importance is not shown).



**Figure 13 – Example: GRL actor that contains a goal and a task**

### 7.3.2 IntentionalElementType

An intentional element can be a Goal, Softgoal, Task, Resource or Belief (see Figure 10).

a) *Abstract grammar*

**i) Attributes**

- None (enumeration metaclass).

**ii) Relationships**

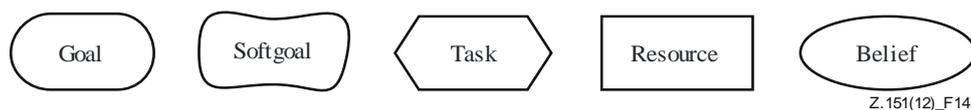
- Used by *IntentionalElement*.

**iii) Constraints**

- None.

b) *Concrete grammar*

The symbols in Figure 14 are used to denote the various types of GRL intentional elements. See usage in *IntentionalElementRef*, clause 7.8.5.



**Figure 14 – Symbol: GRL intentional element types**

c) *Semantics*

- A (hard) Goal is a condition or state of affairs in the world that the stakeholders would like to achieve. How the goal is to be achieved is not specified, allowing alternatives to be considered. A goal can be either a business goal or a system goal. A business goal expresses goals regarding the business or state of the business affairs the individual or organization wishes to achieve. A system goal expresses goals the target system should achieve and generally describes the functional requirements of the target information system.

- A **Softgoal** is a condition or state of affairs in the world that the actor would like to achieve, but unlike in the concept of (hard) goal, there are no clear-cut criteria for whether the condition is achieved, and it is up to subjective judgement and interpretation of the modeller to judge whether a particular state of affairs in fact achieves sufficiently the stated softgoal. Softgoals are often used to describe qualities and non-functional aspects such as security, robustness, performance, usability, etc.
- A **Task** specifies a particular way of doing something. When a task is part of the decomposition of a (higher-level) task, this restricts the higher-level task to that particular course of action. Tasks can also be seen as the solutions in the target system, which will address (or operationalize) goals and softgoals. These solutions provide operations, processes, data representations, structuring, constraints and agents in the target system to meet the needs stated in the goals and softgoals.
- A **Resource** is a physical or informational entity, for which the main concern is whether it is available.
- A **Belief** is used to represent design rationale. Beliefs make it possible for domain characteristics to be considered and properly reflected in the decision-making process, hence facilitating later review, justification and change of the system, as well as enhancing traceability.

## 7.4 GRL links

### 7.4.1 ElementLink

An *ElementLink* connects two GRL linkable elements and represents the intentional relationship existing between them. *ElementLink* abstracts the commonalities of decomposition, contribution and dependency links (see Figure 10).

#### a) Abstract grammar

##### i) Attributes

- Inherits attributes from *GRLmodelElement* (see Figure 9).

##### ii) Relationships

- Inherits relationships from *GRLmodelElement*.
- Contained by *GRLspec* (1): An *ElementLink* is contained in the GRL specification (see Figure 8).
- Association with *GRLLinkableElement* (src, 1): An *ElementLink* has one source GRL linkable element.
- Association with *GRLLinkableElement* (dest, 1): An *ElementLink* has one destination GRL linkable element.
- *ElementLink* is a superclass of *Contribution*, *Dependency* and *Decomposition*.

##### iii) Constraints

- Inherits constraints from *GRLmodelElement*.
- All instances of *ElementLink* shall appear in one of its subclasses (that is, metaclass *ElementLink* is abstract).
- The source and destination GRL linkable elements shall be different.

#### b) Concrete grammar

The concrete syntax for *ElementLink* is further defined in its subclasses.

##### i) Relationships

- Inherits relationships from *GRLmodelElement* (see Figure 30).

- Association with *LinkRef* (0..\*): An *ElementLink* may have link references (see Figure 30).

c) *Semantics*

An *ElementLink* is a directed link that connects a source actor definition, intentional element or indicator to a different destination actor definition, intentional element or indicator. The semantics of the link is provided by the subclass used.

#### 7.4.2 Contribution

A *Contribution* link describes how a source intentional element or source indicator contributes to the satisfaction of a destination intentional element. A contribution is an effect that is a primary desire during modelling, whereas a correlation expresses knowledge about interactions between intentional elements in different categories. A correlation link is the same as a contribution link except that the correlation is not an explicit desire, but is a side-effect and that correlations are only used with intentional elements and not with indicators (see Figure 10).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *ElementLink*.
- contribution (*ContributionType*): The qualitative level of contribution. Default value is Unknown.
- quantitativeContribution (*Integer*): The quantitative level of contribution. Default value is 0 (see clause 9.2.2).
- correlation (Boolean): Indicates whether the link is a regular contribution (false) or a correlation (true). Default value is false.

**ii) Relationships**

- Inherits relationships from *ElementLink*.
- Uses *ContributionType* enumeration.

**iii) Constraints**

- Inherits constraints from *ElementLink*.
- Actor definitions can neither be the source nor the destination of a contribution.
- The destination linkable element (dest) shall not be an intentional element of type Resource or Belief.
- The destination linkable element (dest) shall not be an indicator.
- If the source linkable element (src) is an indicator, then correlation is false.
- quantitativeContribution  $\geq -100$  and quantitativeContribution  $\leq 100$ .

b) *Concrete grammar*

A *Contribution* does not have a visual representation, but link references (*LinkRef*) in GRL diagrams do provide a graphical representation.

**i) Relationships**

- Inherits relationships from *ElementLink*.

c) *Semantics*

A *Contribution* defines the level of impact that the satisfaction of a source intentional element or indicator has on the satisfaction of a destination intentional element. If the impact is qualitative (positive or negative, sufficient or insufficient; see the contributions in clause 7.4.3), then contribution will be used in goal model evaluations. The impact can be

also quantitative (value in  $[-100, 100]$ ) in which case `quantitativeContribution` will be used in goal model evaluations. A correlation link (`correlation` is true) has the same impact on an evaluation as regular contribution links, but it emphasizes side-effects between intentional elements in different categories or actor definitions. Correlations are not used with indicators.

Only the relevant contribution attribute is considered depending on the type of analysis (qualitative or quantitative). It is not required for contribution and `quantitativeContribution` to be consistent, as modellers may want to use only one type of analysis (qualitative or quantitative). However, it is recommended to keep them consistent if the modellers intend to switch between different types of analysis.

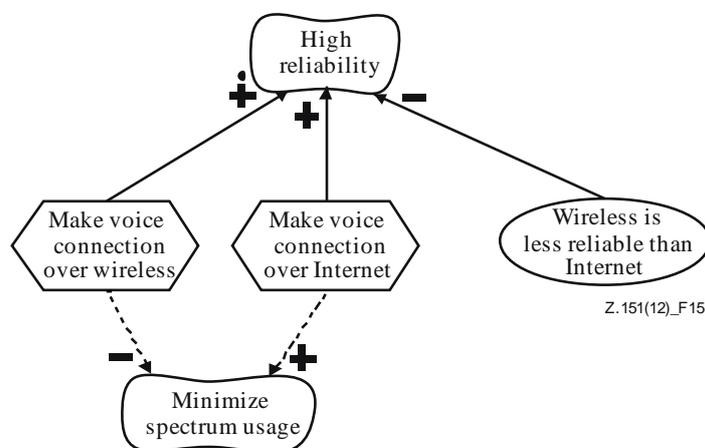
**NOTE – Semantic variation:** Modellers are allowed to impose additional stylistic constraints on the well-formedness of contributions. For example, as it is possible for contribution links to be qualitative and partial, while goals, tasks and resources represent clear-cut concepts, a constraint is able to specify that intentional elements only contribute to softgoals, i.e. the destination intentional element has to be a softgoal.

d) *Model*

None.

e) *Example*

Figure 15 is a GRL diagram that shows three contributions and two correlations linking five intentional elements.



**Figure 15 – Example: GRL contributions and correlations**

- "Make Voice Connection Over Wireless" has a positive and sufficient contribution on "High Reliability".
- "Make Voice Connection Over Internet" has some positive contribution on "High Reliability".
- "Wireless is less reliable than Internet" has some negative contribution on "High Reliability".
- "Make Voice Connection Over Wireless" has some negative correlation (side-effect) on "Minimize Spectrum Usage".
- "Make Voice Connection Over Internet" has some positive correlation (side-effect) on "Minimize Spectrum Usage".

See clauses 7.4.3 and 7.8.7 for the details of the concrete syntax.

### 7.4.3 ContributionType

A qualitative contribution level in a *Contribution* link can take one of the following values: Make, Help, SomePositive, Unknown, SomeNegative, Hurt, Break (see Figure 10).

a) *Abstract grammar*

i) **Attributes**

- None (enumeration metaclass).

ii) **Relationships**

- Used by *Contribution* and by *ContributionChange* (see Figure 29).

iii) **Constraints**

- None.

b) *Concrete grammar*

Figure 16 lists the icons used to annotate GRL contribution links (including correlation links) according to the value of their (qualitative) contribution. See usage in *LinkRef*, clause 7.8.7.



**Figure 16 – Symbol: GRL contribution types**

c) *Semantics*

The qualitative contribution of a source intentional element or indicator to a destination intentional element can be one of the following values based on the degree (positive or negative) and sufficiency of the contribution to the satisfaction of the destination intentional element:

- **Make:** The contribution is positive and sufficient.
- **Help:** The contribution is positive but not sufficient.
- **SomePositive:** The contribution is positive, but the extent of the contribution is unknown.
- **Unknown:** There is some contribution, but the extent and the degree (positive or negative) of the contribution is unknown.
- **SomeNegative:** The contribution is negative, but the extent of the contribution is unknown.
- **Break:** The contribution of the contributing element is negative and sufficient.
- **Hurt:** The contribution is negative but not sufficient.

d) *Model*

An alternative presentation of the Unknown contribution is to simply omit the Unknown icon on the contribution link. This makes GRL diagrams less cluttered, without loss of information.

#### 7.4.4 Dependency

A *Dependency* describes how a source actor definition (the *depender*) depends on a destination actor definition (the *dependee*) for an intentional element or indicator (the *dependum*). Often, the modeller will use two consecutive dependency links (from depender to dependum, and from dependum to dependee) to express detailed dependencies, but dependencies can be used in more generic situations as well (see Figure 10).

The dependum specifies *what* the dependency is about, i.e., the intentional element or indicator around which a dependency relationship centres. With an intentional element or indicator as a source of the dependency, the depender may specify *why* it depends on the dependee for the dependum. With

an intentional element or indicator as a target of the dependency, the dependee may specify *how* it is required to provide or satisfy the dependum.

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *ElementLink*.

**ii) Relationships**

- Inherits relationships from *ElementLink*.

**iii) Constraints**

- Inherits constraints from *ElementLink*.
- Intentional elements of type *Belief* can neither be the source nor the destination of a dependency.
- At least one of the GRL linkable elements linked by a dependency link shall be an actor definition or an intentional element contained in an actor definition or an indicator contained in an actor definition.
- If the source and destination linked by a dependency link are intentional elements or indicators, then these intentional elements and indicators shall not be contained in the same actor definition.

b) *Concrete grammar*

A *Dependency* does not have a visual representation, but link references (*LinkRef*) in GRL diagrams do provide a graphical representation for dependencies.

**i) Relationships**

- Inherits relationships from *ElementLink*.

c) *Semantics*

Dependencies enable reasoning about how actor definitions depend on each other to achieve their goals. The satisfaction level of the depender may be limited by the ability of the dependee to provide the dependum to the depender.

*Dependency* links can be used in a number of configurations including but not limited to the ones described below. According to the required level of detail, intentional elements or indicators inside actor definitions can be used as source and/or destination of a dependency link. Assume *Depender* and *Dependee* are different instances of *Actor* definition, *D1* and *D2* are different instances of *Dependency*, and *Why*, *How* and *What* are different instances of *GRLContainableElement*. *Why* is inside *Depender*, *How* is inside *Dependee* and *What* is not inside any actor definition. A named arrow ( $\text{---}D\text{---}$ ) indicates the presence of a dependency link *D* between the source and target GRL linkable elements involved.

i) *Depender*  $\text{---}D1\text{---}$  *What*  $\text{---}D2\text{---}$  *Dependee*

- *Depender* depends on *Dependee* for *What*. *What* represents the dependum.

ii) *Depender*  $\text{---}D1\text{---}$  *How*

- *Depender* depends on *Dependee* for *How*. The dependum is unknown.

iii) *Why*  $\text{---}D1\text{---}$  *What*  $\text{---}D2\text{---}$  *Dependee*

- *Why* in *Depender* depends on *Dependee* for *What*. *What* represents the dependum.

iv) *Why*  $\text{---}D1\text{---}$  *How*

- *Why* in *Depender* depends on *Dependee* for *How*. The dependum is unknown.

v) *Depender*  $\text{---}D1\text{---}$  *Dependee*

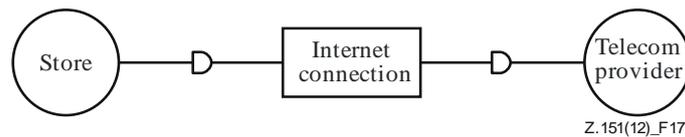
- *Depender* depends on *Dependee*. The dependum is unknown.
- vi) *Why*—*D1*→ *What*—*D2*→ *How*
- *Why* in *Depender* depends on *How* in *Dependee* for *What*. *What* represents the dependum.

d) *Model*

None.

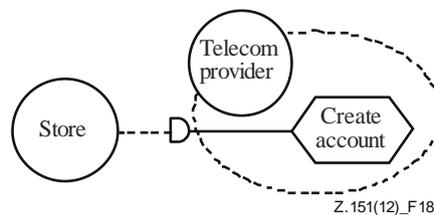
e) *Examples*

The following examples are GRL diagrams illustrating the six configurations discussed in the semantics clause. The same numbering scheme is used. Explanations follow each diagram. The types of intentional elements used here are simply examples and do not preclude other usages in dependencies.



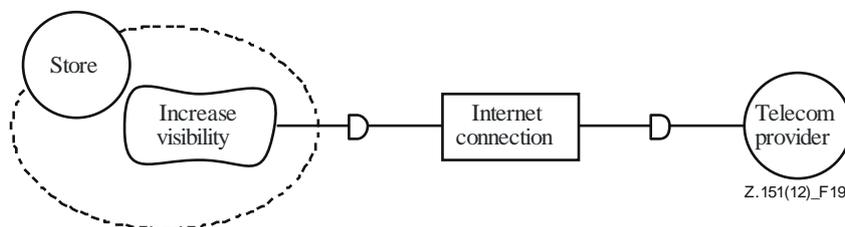
**Figure 17 – Example: GRL dependencies (configuration 1)**

- i) *The Store depends on the Telecom Provider to provide an Internet Connection* (Figure 17). This is a configuration that focuses solely on strategic dependencies between actors. Why and how the dependum is provided are unknown.



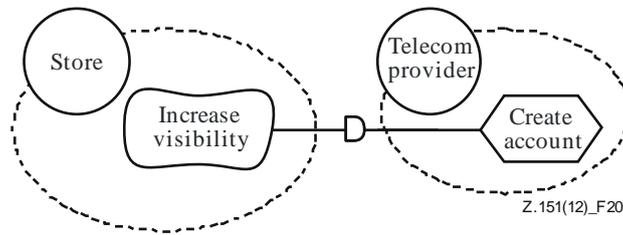
**Figure 18 – Example: GRL dependencies (configuration 2)**

- ii) *The Store depends on the Telecom Provider to create an account* (Figure 18). The dependum and why it is required are unknown.



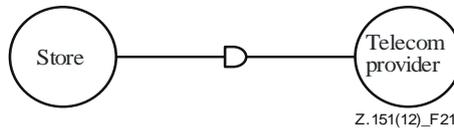
**Figure 19 – Example: GRL dependencies (configuration 3)**

- iii) *To increase its visibility, the Store depends on the Telecom Provider to provide an Internet Connection* (Figure 19). How the dependum is provided is unknown.



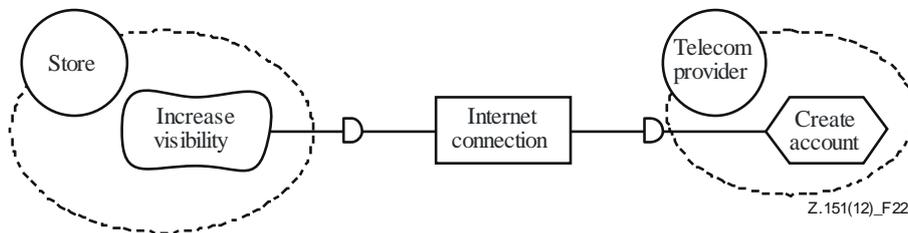
**Figure 20 – Example: GRL dependencies (configuration 4)**

- iv) *To increase its visibility, the Store depends on the Telecom Provider to create an account* (Figure 20). The dependum is unknown.



**Figure 21 – Example: GRL dependencies (configuration 5)**

- v) *The Store depends on the Telecom Provider* (Figure 21). This is a configuration that is typical of preliminary goal models that require further refinement. A dependency is identified, but what, why and how are still unknown.



**Figure 22 – Example: GRL dependencies (configuration 6)**

- vi) *To increase its visibility, the Store depends on the Telecom Provider to provide an Internet Connection by creating an account* (Figure 22). This is a configuration that details the dependum (the Internet connection) together with why it is required and how it is provided.

### 7.4.5 Decomposition

*Decomposition* links provide the ability to define what source intentional elements need to be satisfied or available in order for a target intentional element to be satisfied. The type of decomposition (AND, XOR, IOR) is specified by the *decompositionType* attribute of the target intentional element. Therefore, an intentional element can be decomposed using one decomposition type only (see Figure 10).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *ElementLink*.

**ii) Relationships**

- Inherits relationships from *ElementLink*.

**iii) Constraints**

- Inherits constraints from *ElementLink*.
- Actor definitions can neither be the source nor the destination of a decomposition.

- Intentional elements of type *Belief* can neither be the source nor the destination of a decomposition.
- *Indicators* cannot be the destination of a decomposition.

b) *Concrete grammar*

A *Decomposition* does not have a visual representation, but link references (*LinkRef*) in GRL diagrams do provide a graphical representation.

i) **Relationships**

- Inherits relationships from *ElementLink*.

c) *Semantics*

*Decomposition* links connect the essential parts of an intentional element, which include subtasks that shall be performed, subgoals that shall be achieved, resources that shall be accessible and softgoals that shall be satisfied. There is no ordering between the decomposing elements.

A *Decomposition* link enables the hierarchical decomposition (AND) of a target intentional element by a source element. A target intentional element can be decomposed into many source intentional elements using as many decomposition links. All of the source intentional elements are necessary for the target intentional element to be satisfied.

A *Decomposition* link also enables the description of alternative means of satisfying a target intentional element (XOR for mutually exclusive alternatives, or IOR for alternatives that are not mutually exclusive). One of the source intentional elements is sufficient for the target intentional element to be satisfied.

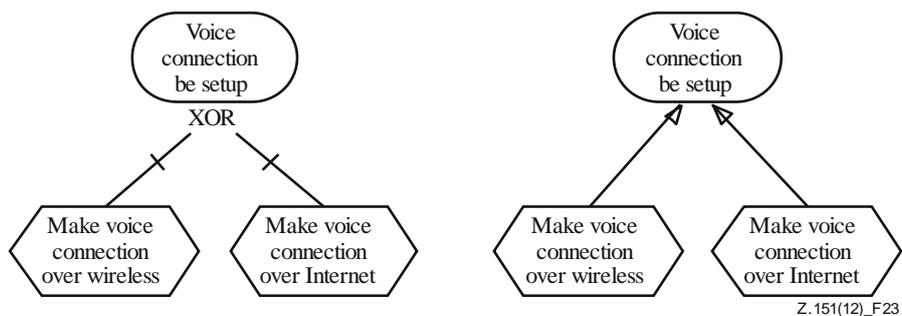
NOTE – **Semantic variation:** Modellers are allowed to impose additional stylistic constraints on the well-formedness of decomposition links. For example, tasks could be limited to AND decomposition, so that if the target of a decomposition link is an intentional element of type *Task*, then its decomposition type has to be AND.

d) *Model*

None.

e) *Examples*

Figure 23 shows two representations of XOR decompositions. On the left, the goal is decomposed into two mutually exclusive alternatives presented as tasks. On the right, the same decomposition is presented using means-end relationships (with the same meaning). See clauses 7.4.6 and 7.8.7 for the details of the concrete syntax.



**Figure 23 – Example: GRL XOR decomposition: normal (left) and means-end (right) presentations**

## 7.4.6 DecompositionType

### a) *Abstract grammar*

An intentional element can be decomposed in one of three ways according to its decompositionType attribute: AND, XOR or IOR (see Figure 10).

#### i) **Attributes**

- None (enumeration metaclass).

#### ii) **Relationships**

- Used by *IntentionalElement*.

#### iii) **Constraints**

None.

### b) *Concrete grammar*

There is no specific icon for decomposition types. The name of the decomposition type itself (AND, XOR or IOR) is used. See usage in *LinkRef*, clause 7.8.7.

### c) *Semantics*

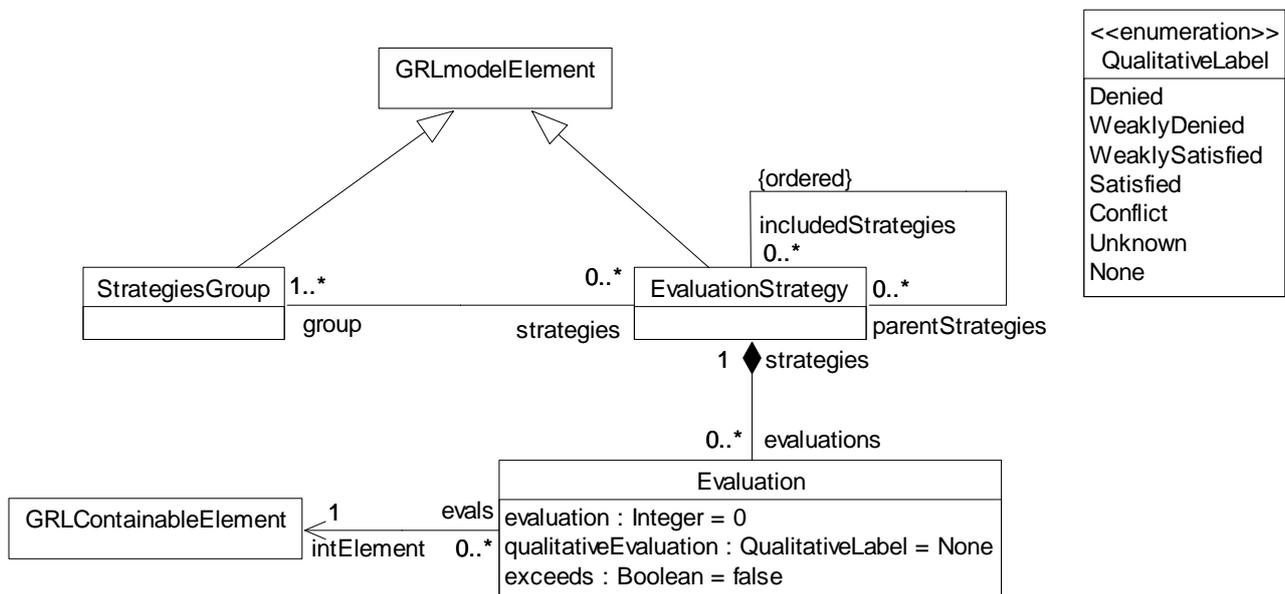
- AND decomposition: The satisfaction of each of the sub-intentional elements is necessary to achieve the target.
- XOR decomposition: The satisfaction of one and only one of the sub-intentional elements is necessary to achieve the target.
- IOR decomposition: The satisfaction of one of the sub-intentional elements is sufficient to achieve the target, but many sub-intentional elements can be satisfied.

## 7.5 GRL strategies

GRL strategies are sets of initial evaluation values given to some containable elements (i.e., intentional elements and indicators) in a GRL model. For indicators, the initial evaluation values are based on real-world measures. These evaluation values, which can be quantitative or qualitative, are satisfaction levels that can then be propagated to the other intentional elements and indicators in the GRL model through the various decomposition, contribution and dependency links connecting them. Evaluations are used to assess how well goals in a model are achieved in a given context, which enables the selection of alternatives that represent appropriate trade-offs among the often-conflicting goals of the stakeholders/actors involved. A good strategy provides rationale and documentation for decisions leading to requirements, providing better context for standards/system developers and implementers while avoiding unnecessary re-evaluations of worse alternative strategies.

NOTE 1 – The name *intElement* is used for backward compatibility with previous versions of this Recommendation. A more appropriate name would be *contElement*.

NOTE 2 – The name *group* is used for backward compatibility with previous versions of this Recommendation. A more appropriate name would be *groups*.



**Figure 24 – Abstract grammar: GRL evaluation strategies**

### 7.5.1 StrategiesGroup

A *StrategiesGroup* is a collection of evaluation strategies. It is used to organize evaluation strategies and to manipulate them as a group (see Figure 24).

#### a) Abstract grammar

##### i) Attributes

- Inherits attributes from *GRLmodelElement*.

##### ii) Relationships

- Inherits relationships from *GRLmodelElement*.
- Contained by *GRLspec* (1): A *StrategiesGroup* is contained in the GRL specification (see Figure 8).
- Association with *EvaluationStrategy* (0..\*): A *StrategiesGroup* may refer to evaluation strategies.

##### iii) Constraints

- Inherits constraints from *GRLmodelElement*.

#### b) Concrete grammar

A *StrategiesGroup* does not have a graphical visual representation. Optionally, a *StrategiesGroup* is visualized using the textual syntax for *StrategiesGroup* as specified in Annex B.

##### i) Relationships

- Inherits relationships from *GRLmodelElement*.

#### c) Semantics

None (*StrategiesGroup* is a structural concept only).

### 7.5.2 EvaluationStrategy

An *EvaluationStrategy* is a collection of evaluations. It is used to define satisfaction levels for a subset of the intentional elements of a GRL specification. An evaluation strategy provides the initial context for GRL model analysis based on a satisfaction propagation algorithm. The same evaluation strategy may be part of multiple groups of strategies (see Figure 24).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *GRLmodelElement*.

**ii) Relationships**

- Inherits relationships from *GRLmodelElement*.
- Contained by *GRLspec* (1): An *EvaluationStrategy* is contained in the GRL specification (see Figure 8).
- Composition of *Evaluation* (0..\*): An *EvaluationStrategy* may contain evaluations.
- Association with *StrategiesGroup* (1..\*): An *EvaluationStrategy* is referenced by at least one group of strategies.
- Association with *EvaluationStrategy* (parentStrategies, 0..\*): An *EvaluationStrategy* may be included by evaluation strategies.
- Association with *EvaluationStrategy* (includedStrategies, 0..\*) {ordered}: An *EvaluationStrategy* may have an ordered collection of included evaluation strategies.

**iii) Constraints**

- Inherits constraints from *GRLmodelElement*.
- The strategy containment hierarchy established by the includedStrategies relationship does not contain any cycles (i.e., an *EvaluationStrategy* shall not appear more than once on a path from a top node to a leaf node in the containment hierarchy).
- Let LE be the list of *Evaluations* of an *EvaluationStrategy*. Let LCE be the list of *GRLContainableElements* associated with at least one *Evaluation* in LE. LCE shall not contain the same *GRLContainableElement* more than once.
- Let LE be the list of *Evaluations* of an *EvaluationStrategy* and the list of *Evaluations* of all includedStrategies of the same *EvaluationStrategy*. Let LEI be the subset of LE that contains only those *Evaluations* that are associated with a *GRLContainableElement* of type *Indicator*. Let LI be the list of *Indicators* associated with at least one *Evaluation* in LEI. For each *Indicator* in LI, there exists at least one *Evaluation* in LEI that is associated with the indicator and that is also associated with an *IndicatorConversion*. For each *Indicator* in LI, there exists at least one *Evaluation* in LEI that is associated with the indicator and that also contains an *IndicatorEvaluation*.

b) *Concrete grammar*

An *EvaluationStrategy* does not have a graphical visual representation. Optionally, an *EvaluationStrategy* is visualized using the textual syntax for *EvaluationStrategy* as specified in Annex B.

**i) Relationships**

- Inherits relationships from *GRLmodelElement*.

c) *Semantics*

The *Evaluations* contained in an *EvaluationStrategy* represent an initial context for the evaluation of a GRL model. Using a model evaluation algorithm (see clause 11.1), the initial values specified in the evaluations are propagated to the intentional elements and indicators that do not have any initial evaluation value, through the element links that connect them.

If an evaluation strategy contains another evaluation strategy, the evaluation algorithm considers the union of the *Evaluations*. For conflicting *Evaluations* (i.e., evaluations of the same containable element) however, the *Evaluation* of the containing evaluation strategy overrides the *Evaluation* of the contained evaluation strategy. If more than one evaluation strategy is included, then the last evaluation strategy in the ordered list of included evaluation

strategies takes precedence over the previous ones (i.e., the evaluations are applied beginning with the first included evaluation strategy in the ordered list; then each other evaluation strategy is applied up to the last included evaluation strategy, and finally the evaluations of the including evaluation strategy are applied).

### 7.5.3 Evaluation

An *Evaluation* provides initial quantitative and qualitative evaluation values to a containable element (see Figure 24).

#### a) *Abstract grammar*

##### i) **Attributes**

- *evaluation* (*Integer*): Initial quantitative satisfaction value (also called evaluation value) of the associated intentional element. Default value is 0 (see clause 9.2.2).
- *qualitativeEvaluation* (*QualitativeLabel*): Initial qualitative satisfaction value (also called evaluation value) of the associated intentional element. Default value is None.
- *exceeds* (*Boolean*): Indicates whether the quantitative and qualitative evaluations exceed expectations (*true*) or not (*false*). Default value is *false*.

##### ii) **Relationships**

- Contained by *EvaluationStrategy* (1): An *Evaluation* is contained in one evaluation strategy.
- Composition of *IndicatorEvaluation* (0..1): An *Evaluation* may contain one set of real-world values (see Figure 26).
- Association with *GRLContainableElement* (1): An *Evaluation* provides initial evaluation values to one containable element.
- Association with *IndicatorConversion* (0..1): An *Evaluation* may have one conversion method to convert real-world values into GRL evaluation values (see Figure 26).
- Uses *QualitativeLabel* enumeration.

##### iii) **Constraints**

- $\text{evaluation} \geq -100$  and  $\text{evaluation} \leq 100$ .
- If *exceeds* is *true*, then *evaluation* is 100 and *qualitativeEvaluation* is *Satisfied*.
- If the *intElement* of an *Evaluation* is of type *Indicator*, then the conversion shall be specified for the *Evaluation*.
- If the *intElement* of an *Evaluation* is of type *Indicator*, then the unit of the *Indicator* of the *Evaluation* is the same as the unit of the *IndicatorConversion* of the *Evaluation*.

#### b) *Concrete grammar*

An *Evaluation* does not have a visual representation. However, it may impact the presentation of intentional (and indicator) element references (see clause 7.8.5).

#### c) *Semantics*

An *Evaluation* defines the initial level of satisfaction of an intentional element and indicator. In the case of indicators, the initial level of satisfaction is based on a real-world measure (*IndicatorEvaluation*) and a conversion method (*IndicatorConversion*). If the level of satisfaction is qualitative (see the qualitative label types in clause 7.5.4), then *qualitativeEvaluation* will be used in goal model evaluations. If the level of satisfaction is quantitative (integer value between  $-100$  for sufficiently denied and  $+100$  for sufficiently satisfied, inclusively), then *evaluation* will be used in goal model evaluations. An evaluation value of 0 means that the intentional element is neither satisfied nor denied.

The *exceeds* attribute captures the fact that the initial level of satisfaction goes beyond expectations. This attribute is mostly used with indicators to indicate that the measured real-world value actually exceeds the agreed upon value which, when converted into a GRL evaluation value, results in full satisfaction.

Only the relevant evaluation attribute is considered depending on the type of analysis (qualitative or quantitative). It is not required for evaluation and qualitativeEvaluation to be consistent as modellers may want to use only one type of analysis (qualitative or quantitative). However, it is recommended to keep them consistent if the modellers intend to switch between different types of analysis.

#### 7.5.4 QualitativeLabel

A *QualitativeLabel* represents the qualitative satisfaction level of an intentional element, indicator or actor. It is also used by conversions from qualitative real-world values to GRL evaluation values. It can be one of the following values: Denied, WeaklyDenied, WeaklySatisfied, Satisfied, Conflict, Unknown and None (see Figure 24).

a) *Abstract grammar*

##### i) Attributes

– None (enumeration metaclass).

##### ii) Relationships

– Used by *Evaluation* and *QualToQMapping* (see Figure 26).

##### iii) Constraints

None.

b) *Concrete grammar*

Figure 25 lists the icons that are used to annotate GRL intentional elements, indicators and actors according to their (qualitative) satisfaction level for a given strategy evaluation. See usage in *ActorRef*, clause 7.8.3, *IntentionalElementRef*, clause 7.8.5, *CollapsedActorRef*, clause 7.8.6.



**Figure 25 – Symbol: GRL qualitative labels**

c) *Semantics*

The qualitative satisfaction level of an intentional element or indicator can be one of the following values based on the degree (positive or negative) and magnitude of the satisfaction:

- Denied: The intentional element or indicator is sufficiently dissatisfied.
- WeaklyDenied: The intentional element or indicator is partially dissatisfied.
- WeaklySatisfied: The intentional element or indicator is partially satisfied.
- Satisfied: The intentional element or indicator is sufficiently satisfied.
- Conflict: There are arguments strongly in favour and strongly against the satisfaction of the intentional element or indicator.
- Unknown: The satisfaction level of the intentional element or indicator is unknown.
- None: The intentional element or indicator is neither satisfied nor dissatisfied.

d) *Model*

An alternative presentation of the None satisfaction level is to simply omit the None icon on the intentional element or indicator. This makes GRL diagrams less crowded, without loss of information.

## 7.6 GRL indicators

GRL indicators represent real-world measurements in the GRL model. Two types of conversions translate a quantitative or qualitative real-world measurement into a quantitative or qualitative GRL evaluation value which can then be taken into consideration during GRL model analysis.

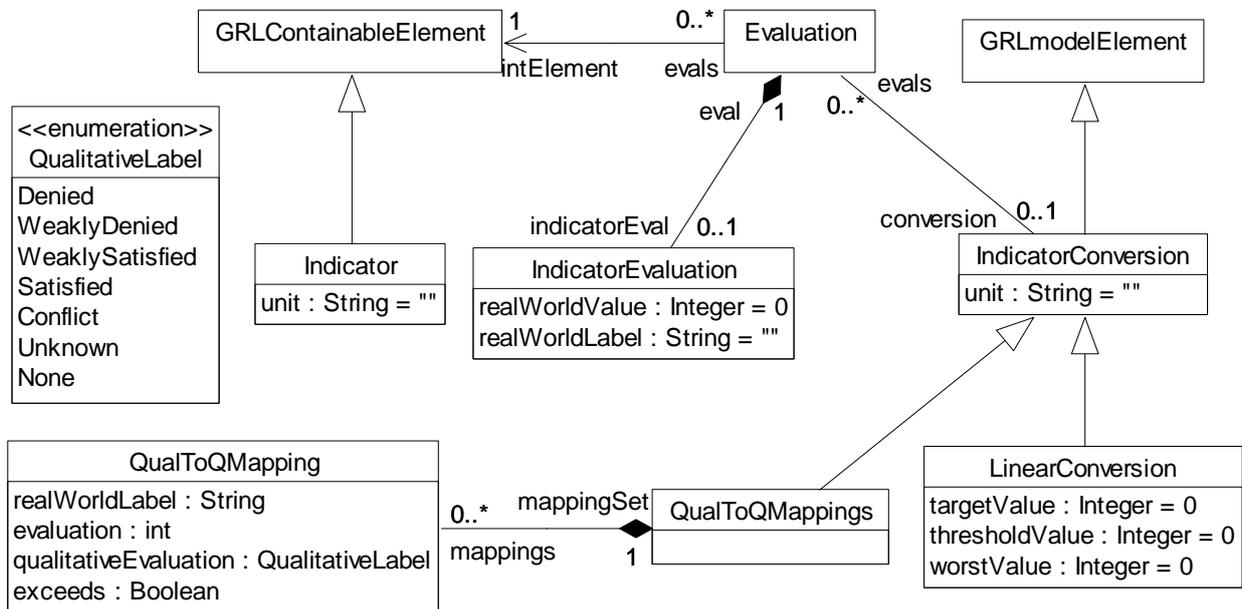


Figure 26 – Abstract grammar: GRL indicators

### 7.6.1 Indicator

An *Indicator* is a GRL containable element used for models that base GRL model analysis on real-world measurements. An indicator substantiates qualitative or quantitative evaluation values in the GRL model with the help of real-world data, allowing GRL model analysis to be more accurate and realistic than with satisfaction values only. Indicators may be included in actor definitions, they can be linked to each other in different ways and they can be given a quantitative or qualitative importance level when included in an actor definition (see Figure 26).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *GRLContainableElement*.
- unit (String): Indicates the measurement unit of the real-world values. Default value is the empty String "".

**ii) Relationships**

- Inherits relationships from *GRLContainableElement*.

**iii) Constraints**

- Inherits constraints from *GRLContainableElement*.
- If an *Indicator* is associated with an *Actor* definition, then there is only one *Indicator* with this name associated with the *Actor* definition.

- If an *Indicator* is not associated with any *Actor* definition, then there is only one *Indicator* with this name that is not associated with any *Actor* definition.

b) *Concrete grammar*

An indicator does not have a visual representation, but intentional element (or indicator) references (*IntentionalElementRef*) in GRL diagrams do have a graphical representation.

**i) Relationships**

- Inherits relationships from *GRLContainableElement*.

c) *Semantics*

An *Indicator* describes a qualitative or quantitative real-world measurement (*IndicatorEvaluation* contained by *Evaluation* associated with superclass *GRLContainableElement* of the indicator). This measurement is converted into a qualitative or quantitative GRL evaluation value (*Evaluation*) as defined by a conversion method (*IndicatorConversion* associated with the same *Evaluation*). An *Indicator* may have several *IndicatorEvaluations*, i.e., one for each *EvaluationStrategy*.

An indicator contained in an *Actor* definition belongs to this actor definition and therefore describes an actor-specific measurement.

The unit attribute of *Indicator* defines the unit of the real-world measurements of the indicator as defined in the previous paragraph (i.e., *realWorldValue* or *realWorldLabel* of *IndicatorEvaluation*). If decimal points are desired for the *realWorldValue*, then the unit is adjusted as the *realWorldValue* is always an integer value. For example, to set the *realWorldValue* to \$1.15, the *realWorldValue* is set to 115 and the unit of the *Indicator* is set to "cents". Similarly, to set the *realWorldValue* to 3.5%, the *realWorldValue* is set to 35 and the unit is set to "1/10 of a percent".

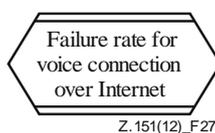
The unit of *Indicator* and unit of all its *IndicatorConversions* applied to the real-world measurements of the indicator (i.e., the *realWorldValues* or *realWorldLabels* of all *IndicatorEvaluations*) shall be the same.

d) *Model*

None.

e) *Examples*

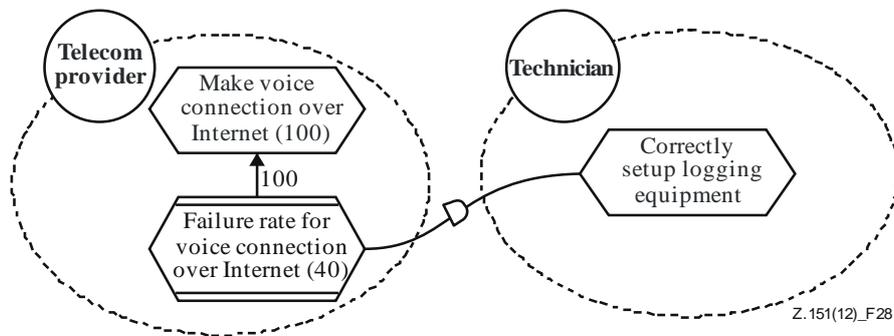
Figure 27 is a GRL diagram with one indicator which captures the failure rate of voice connections over the Internet.



**Figure 27 – Example: GRL indicator**

See clauses 7.1.4 and 7.8.5 for the details of the concrete syntax.

The GRL diagram in Figure 28 shows a task and an indicator contained by a "Telecom Provider" actor. The indicator has an importance value of 40 while the task has an importance value of 100. The *Contribution* link between the indicator and the task has a quantitativeContribution of 100. The indicator also has a *Dependency* link with a task in a "Technician" actor.



**Figure 28 – Example: GRL model with an indicator**

### 7.6.2 IndicatorEvaluation

An *IndicatorEvaluation* provides initial quantitative and qualitative real-world values to a containable element of type *Indicator* (see Figure 26).

a) *Abstract grammar*

#### i) Attributes

- *realWorldValue* (*Integer*): Initial quantitative real-world value of the associated indicator. Default value is 0 (see clause 9.2.2).
- *realWorldLabel* (*String*): Initial qualitative real-world value of the associated indicator. Default value is the empty String "".

#### ii) Relationships

- Contained by *Evaluation* (1): An *IndicatorEvaluation* is contained in one evaluation.

#### iii) Constraints

- The *GRLContainableElement* of the *Evaluation* containing an *IndicatorEvaluation* shall be of type *Indicator*.

b) *Concrete grammar*

An *IndicatorEvaluation* does not have a graphical visual representation. However, it may impact the presentation of indicator element references (see clause 7.8.5).

c) *Semantics*

An *IndicatorEvaluation* defines the initial real-world values of an indicator. If the real-world value is qualitative, then *realWorldLabel* will be used when converting the real-world value into GRL evaluations. If the level of satisfaction is quantitative, then *realWorldValue* will be used by the conversion. The conversion method is defined by the *IndicatorConversion* of the *Evaluation* containing the *IndicatorEvaluation*.

Only the relevant real-world value is considered depending on the type of conversion. It is not required for *realWorldValue* and *realWorldLabel* to be consistent as modellers may want to use only one type of conversion. However, it is recommended to keep them consistent if the modellers intend to switch between different types of conversions.

### 7.6.3 IndicatorConversion

An *IndicatorConversion* defines which quantitative and qualitative real-world values to convert into GRL evaluation values (see Figure 26).

a) *Abstract grammar*

#### i) Attributes

- Inherits attributes from *GRLmodelElement* (see Figure 9).

- `unit` (`String`): Indicates the measurement unit of the real-world values. Default value is the empty `String` "".

### ii) Relationships

- Inherits relationships from *GRLmodelElement*.
- Contained by *GRLspec* (1): An *IndicatorConversion* is contained in the GRL specification (see Figure 8).
- Association with *Evaluation* (0..\*): An *IndicatorConversion* may convert the initial real-world values associated with many *Evaluations*.
- *IndicatorConversion* is a superclass of *LinearConversion* and *QualToQMappings*.

### iii) Constraints

- Inherits constraints from *GRLmodelElement*.
- All instances of *IndicatorConversion* shall appear in one of its subclasses (that is, metaclass *IndicatorConversion* is abstract).

## b) Concrete grammar

An *IndicatorConversion* does not have a graphical visual representation. Optionally, concrete syntax for the subclasses of *IndicatorConversion* is defined using the textual syntax for the subclasses as specified in Annex B.

### i) Relationships

- Inherits attributes from *GRLmodelElement* (see Figure 30).

## c) Semantics

The subclasses of *IndicatorConversion* specify how the initial real-world values of an indicator (*IndicatorEvaluation* of the associated *Evaluation*) are converted into GRL evaluation values (*Evaluation*). These subclasses may have additional attributes and relationships as required for the conversion.

An *IndicatorConversion* may only be chosen for an *Indicator*, if the unit of both is the same. For more details on units and decimal points for real-world values, see *Indicator*, clause 7.6.1.

## 7.6.4 LinearConversion

A *LinearConversion* converts quantitative real-world values into quantitative and qualitative GRL evaluation values based on linear intrapolation (see Figure 26).

### a) Abstract grammar

#### i) Attributes

- Inherits attributes from *IndicatorConversion*.
- `targetValue` (*Integer*): Indicates the best result that can be measured for this quantitative real-world value. Default value is 0 (see clause 9.2.2).
- `thresholdValue` (*Integer*): Indicates the measurement result where the quantitative real-world value is neither negative nor positive but neutral. Default value is 0 (see clause 9.2.2).
- `worstValue` (*Integer*): Indicates the worst result that can be measured for this quantitative real-world value. Default value is 0 (see clause 9.2.2).

#### ii) Relationships

- Inherits relationships from *IndicatorConversion*.

#### iii) Constraints

- Inherits constraints from *IndicatorConversion*.

- The three value attributes of a *LinearConversion* are either sorted in ascending or descending order (i.e., either (i) the *targetValue* is greater than or equal to the *thresholdValue* and the *thresholdValue* is greater than or equal to the *worstValue* or (ii) the *targetValue* is less than or equal to the *thresholdValue* and the *thresholdValue* is less than or equal to the *worstValue*).

b) *Concrete grammar*

A *LinearConversion* does not have a graphical visual representation. Optionally, a *LinearConversion* is visualized using the textual syntax for *LinearConversion* as specified in Annex B.

i) **Relationships**

- Inherits relationships from *IndicatorConversion*.

c) *Semantics*

A *LinearConversion* specifies three real-world values, i.e., the *targetValue*, the *thresholdValue* and the *worstValue*, and then maps the *targetValue* to the GRL evaluation value 100, the *thresholdValue* to the GRL evaluation value 0 and the *worstValue* to the GRL evaluation value –100. An *IndicatorEvaluation* defines a quantitative real-world value *R* (*realWorldValue*) that is to be converted by the *LinearConversion* associated with the *IndicatorEvaluation* through an *Evaluation*. A real-world value *R* that falls between the *targetValue* and the *thresholdValue* is intrapolated to the [0, 100] GRL evaluation value range according to the formula:  $(R - \text{thresholdValue}) / (\text{targetValue} - \text{thresholdValue}) \times 100$ . A real-world value *R* that falls between the *thresholdValue* and the *worstValue* is intrapolated to the [-100, 0] GRL evaluation value range according to the formula:  $(R - \text{thresholdValue}) / (\text{worstValue} - \text{thresholdValue}) \times (-100)$ .

If the *targetValue* is the same as the *thresholdValue* or the *worstValue* is the same as the *thresholdValue*, then the result of the conversion is the GRL evaluation value of 0. If *R* is outside the [*targetValue*, *worstValue*] real-world value range, then the result of the conversion is the GRL evaluation value of 100, if *R* is beyond the *targetValue*. In this case, the *exceeds* value is set to true. In all other cases, the *exceeds* value is set to false. If *R* is beyond the *worstValue*, then the result of the conversion is the GRL evaluation value of –100.

NOTE – **Semantic variation:** The GRL *qualitativeEvaluation* value is calculated based on the resulting GRL evaluation value and a mapping from quantitative GRL evaluation values to qualitative GRL evaluation values. One possible mapping is shown in Table 1, but modellers are allowed to define any other mapping deemed appropriate.

**Table 1 – GRL mapping from quantitative to qualitative evaluation values**

<i>Quantitative value</i>	<i>Qualitative value</i>
–100	<i>Denied</i>
(–100, 0)	<i>WeaklyDenied</i>
0	<i>None</i>
(0, 100)	<i>WeaklySatisfied</i>
100	<i>Satisfied</i>

d) *Model*

None.

e) *Examples*

Table 2 gives an example for a linear conversion based on the GRL model in Figure 28, assuming that the unit of measurement for the indicator in Figure 28 is percentage. The linear conversion specifies the *targetValue* as 0%, the *thresholdValue* as 5% and the *worstValue* as 100%, and then maps these values to the GRL evaluation values 100, 0 and -100, respectively. Three sample real-world values and their corresponding GRL evaluation values are also given. The first two use the first formula, while the last one uses the second formula specified in this clause.

**Table 2 – GRL linear conversion**

	<i>Target</i>	<i>Threshold</i>	<i>Worst</i>	<i>Example 1</i>	<i>Example 2</i>	<i>Example 3</i>
<i>Real-world value</i>	0%	5%	100%	1%	3.5%	45%
<i>GRL evaluation value</i>	100	0	-100	$(1-5)/(0-5) \times 100 = 80$	$(3.5-5)/(0-5) \times 100 = 30$	$(45-5)/(100-5) \times (-100) = -42$

### 7.6.5 QualToQMappings

A *QualToQMappings* converts qualitative real-world values into quantitative and qualitative GRL evaluation values based on explicit mappings (see Figure 26).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *IndicatorConversion*.

**ii) Relationships**

- Inherits relationships from *IndicatorConversion*.
- Composition of *QualToQMappings* (0..\*): A *QualToQMappings* conversion may contain many mappings.

**iii) Constraints**

- Inherits constraints from *IndicatorConversion*.

b) *Concrete grammar*

A *QualToQMappings* conversion does not have a graphical visual representation. Optionally, a *QualToQMappings* conversion is visualized using the textual syntax for *QualToQMappings* as specified in Annex B.

**i) Relationships**

- Inherits relationships from *IndicatorConversion*.

c) *Semantics*

None (*QualToQMappings* is a structural concept only).

### 7.6.6 QualToQMapping

A *QualToQMapping* converts a single qualitative real-world value into quantitative and qualitative GRL evaluation values based on explicit mappings (see Figure 26).

a) *Abstract grammar*

**i) Attributes**

- *realWorldLabel* (String): Indicates a qualitative, measurable, real-world value.

- *evaluation* (*Integer*): Indicates the mapped initial quantitative satisfaction value (also called evaluation value) (see clause 9.2.2).
- *qualitativeEvaluation* (*QualitativeLabel*): Indicates the mapped initial qualitative satisfaction value (also called evaluation value).
- *exceeds* (*Boolean*): Indicates whether the mapped values exceed expectations (true) or not (false).

**ii) Relationships**

- Contained by *QualToQMappings* (1): A *QualToQMapping* is contained in one *QualToQMappings* conversion.
- Uses *QualitativeLabel* enumeration.

**iii) Constraints**

- $evaluation \geq -100$  and  $evaluation \leq 100$ .
- If *exceeds* is true, then *evaluation* is 100 and *qualitativeEvaluation* is Satisfied.

**b) Concrete grammar**

A *QualToQMapping* does not have a graphical visual representation. Optionally, a *QualToQMapping* is visualized using the textual syntax for *QualToQMapping* as specified in Annex B.

**c) Semantics**

A *QualToQMapping* specifies one qualitative real-world value (*realWorldLabel*), one qualitative GRL evaluation value (*qualitativeEvaluation*), one quantitative GRL evaluation value (*evaluation*) and the exceeds flag (*exceeds*). An *IndicatorEvaluation* defines a qualitative real-world value R (*realWorldLabel*) that is to be converted by the *QualToQMapping* associated with the *IndicatorEvaluation* through an *Evaluation* E. If R matches the *realWorldLabel* of *QualToQMapping*, then the two GRL evaluation values and the exceeds flag are copied from the *QualToQMapping* to E.

**d) Model**

None.

**e) Examples**

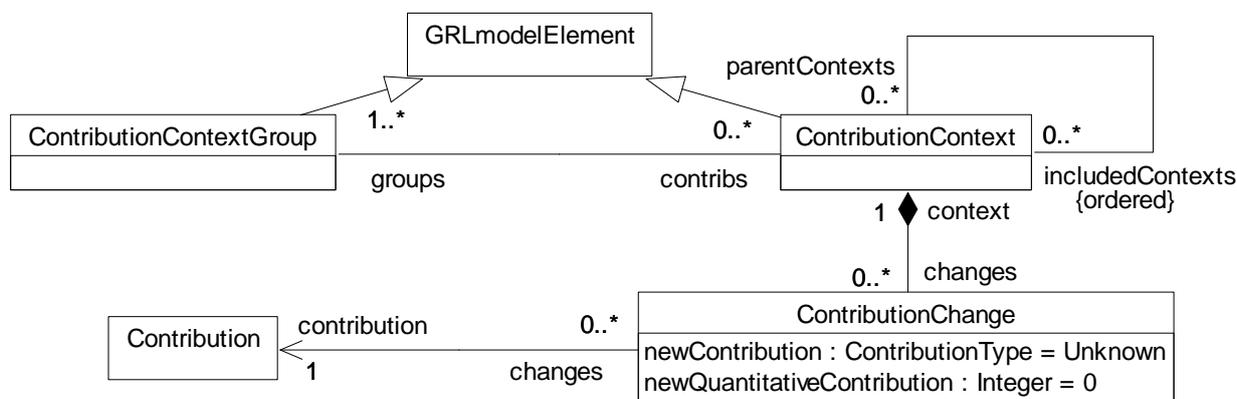
Table 3 gives an example for three *QualToQMappings* based on the GRL model in Figure 28, assuming that the indicator in Figure 28 is measured by classifying the equipment used to establish connections.

**Table 3 – GRL mappings of qualitative real-world value**

<i>realWorldLabel</i>	<i>qualitativeEvaluation</i>	<i>evaluation</i>	<i>exceeds</i>
Class 1	<i>Satisfied</i>	100	false
Class 2	<i>WeaklySatisfied</i>	25	false
Class 3	<i>WeaklyDenied</i>	-25	false

**7.7 GRL contribution contexts**

GRL contribution contexts are sets of changes to quantitative and qualitative contributions in a GRL model. When a contribution context is selected, the changes override the existing contributions and the overridden contributions are subsequently considered by GRL goal model analysis.



**Figure 29 – Abstract grammar: GRL contribution contexts**

### 7.7.1 ContributionContextGroup

A *ContributionContextGroup* is a collection of contribution contexts. It is used to organize sets of contribution changes (also known as overrides) and to manipulate them as a group (see Figure 29).

a) *Abstract grammar*

#### i) Attributes

- Inherits attributes from *GRLmodelElement* (see Figure 9).

#### ii) Relationships

- Inherits relationships from *GRLmodelElement*.
- Contained by *GRLspec* (1): A *ContributionContextGroup* is contained in the GRL specification (see Figure 8).
- Association with *ContributionContext* (0..\*): A *ContributionContextGroup* may refer to contribution contexts.

#### iii) Constraints

- Inherits constraints from *GRLmodelElement*.

b) *Concrete grammar*

A *ContributionContextGroup* does not have a graphical visual representation. Optionally, a *ContributionContextGroup* is visualized using the textual syntax for *ContributionContextGroup* as specified in Annex B.

#### i) Relationships

- Inherits relationships from *GRLmodelElement* (see Figure 30).

c) *Semantics*

None (*ContributionContextGroup* is a structural concept only).

### 7.7.2 ContributionContext

A *ContributionContext* is a collection of contribution changes. The same contribution context may be part of multiple groups of contribution contexts (see Figure 29).

a) *Abstract grammar*

#### i) Attributes

- Inherits attributes from *GRLmodelElement* (see Figure 9).

#### ii) Relationships

- Inherits relationships from *GRLmodelElement*.

- Contained by *GRLspec* (1): A *ContributionContext* is contained in the GRL specification (see Figure 8).
- Composition of *ContributionChange* (0..\*): A *ContributionContext* may contain contribution changes.
- Association with *ContributionContextGroup* (1..\*): A *ContributionContext* is referenced by at least one group of contribution contexts.
- Association with *ContributionContext* (parentContexts, 0..\*): A *ContributionContext* may be included by contribution contexts.
- Association with *ContributionContext* (includedContexts, 0..\*) {ordered}: A *ContributionContext* may have an ordered collection of included contribution contexts.

### iii) Constraints

- Inherits constraints from *GRLmodelElement*.
- The contribution context containment hierarchy established by the includedContexts relationship does not contain any cycles (i.e., a *ContributionContext* shall not appear more than once on a path from a top node to a leaf node in the containment hierarchy).
- Let LC be the list of *ContributionChanges* of a *ContributionContext*. Let LCC be the list of *Contributions* associated with at least one *ContributionChange* in LC. LCC shall not contain the same *Contribution* more than once.

#### b) Concrete grammar

A *ContributionContext* does not have a graphical visual representation. Optionally, a *ContributionContext* is visualized using the textual syntax for *ContributionContext* as specified in Annex B.

#### i) Relationships

- Inherits relationships from *GRLmodelElement* (see Figure 30).

#### c) Semantics

The *ContributionChanges* contained in a *ContributionContext* represent overrides to existing quantitative and qualitative contributions. When a contribution context is selected, the overrides are applied and a model evaluation algorithm (see clause 11.1) that propagates evaluations to intentional elements and indicators through their element links uses the overridden contributions.

When a contribution context is selected that contains another contribution context, the union of *ContributionChanges* are applied. For conflicting *ContributionChanges* (i.e., overrides to the same contribution) however, the *ContributionChange* of the containing contribution context overrides the *ContributionChange* of the contained contribution context. If more than one contribution context is included, then the last contribution context in the ordered list of included contribution contexts takes precedence over the previous ones (i.e., the overrides are applied beginning with the first included contribution context in the ordered list; then each other contribution context is applied up to the last included contribution context, and finally the overrides of the including contribution context are applied).

### 7.7.3 ContributionChange

A *ContributionChange* provides quantitative and qualitative contribution attributes that override an existing contribution (see Figure 29).

a) *Abstract grammar*

**i) Attributes**

- `newContribution` (*ContributionType*): The overriding qualitative contribution level. Default value is `Unknown`.
- `newQuantitativeContribution` (*Integer*): The overriding quantitative contribution level. Default value is 0 (see clause 9.2.2).

**ii) Relationships**

- Contained by *ContributionContext* (1): A *ContributionChange* is contained in one contribution context.
- Association with *Contribution* (1): A *ContributionChange* provides overrides for one contribution.
- Uses *ContributionType* enumeration.

**iii) Constraints**

- $\text{newQuantitativeContribution} \geq -100$  and  $\text{newQuantitativeContribution} \leq 100$ .

b) *Concrete grammar*

A *ContributionChange* does not have a graphical visual representation. However, it may impact the presentation of link references (*LinkRef*) in GRL diagrams (see clause 7.8.7). Optionally, a *ContributionChange* is visualized using the textual syntax for *ContributionChange* as specified in Annex B.

c) *Semantics*

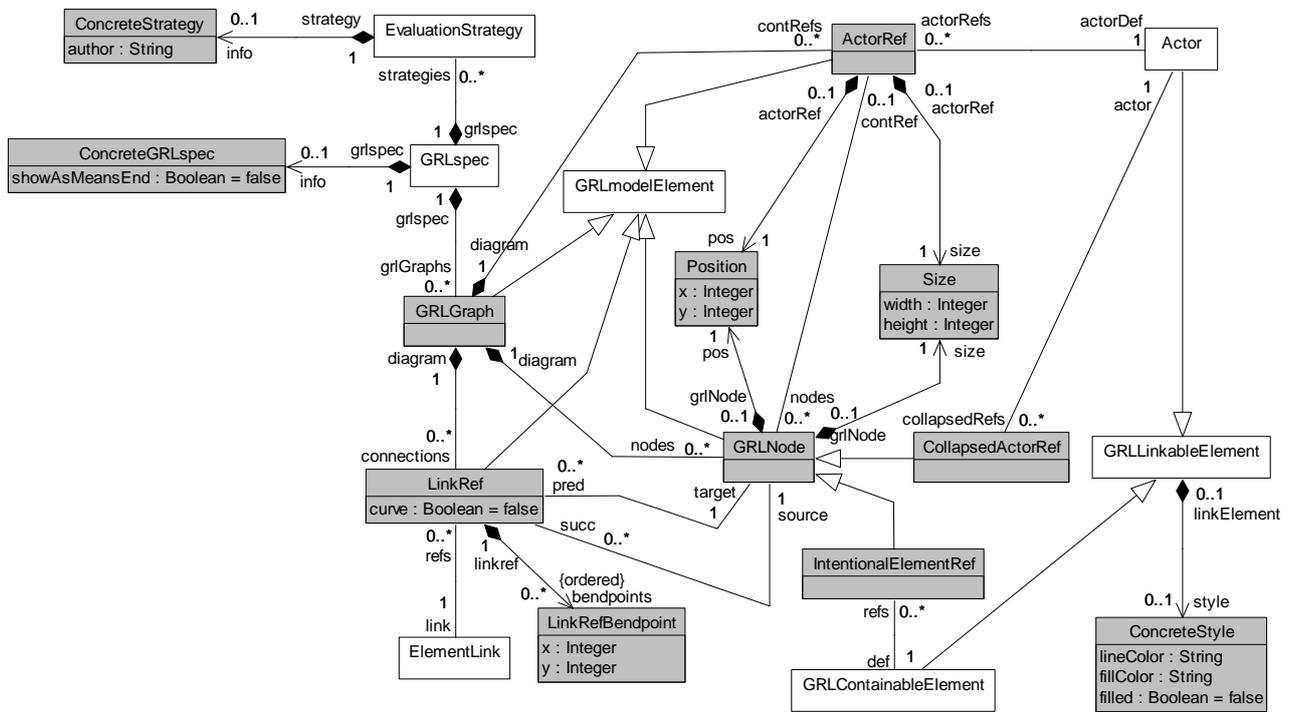
A *ContributionChange* defines overrides for the quantitative and qualitative contribution attributes of an existing *Contribution*. The overrides are applied when the *ContributionContext* of the *Contribution* is selected.

Only the relevant overridden contribution attribute is considered depending on the type of analysis (qualitative or quantitative). It is not required for `newContribution` and `newQuantitativeContribution` to be consistent as modellers may want to use only one type of analysis (qualitative or quantitative). However, it is recommended to keep them consistent if the modellers intend to switch between different types of analysis.

## 7.8 GRL concrete grammar metaclasses

The following concrete grammar metaclasses may be contained by some of the GRL abstract grammar metaclasses. They have no semantics. Figure 30 shows all concrete grammar metaclasses that are GRL specific except for *Label* (see clause 7.8.8). Further details are shown for those concrete grammar metaclasses that are shared between GRL and UCM in clauses 7.8.8, 7.8.10, 7.8.11, 7.8.12 and 7.8.13.

NOTE – The name `IntentionalElementRef` is used for backward compatibility with previous versions of this Recommendation. A more appropriate name would be `ContainableElementRef`.



**Figure 30 – Concrete grammar: GRL concrete syntax metaclasses**

### 7.8.1 ConcreteGRLspec

*ConcreteGRLspec* defines how GRL XOR and IOR *Decomposition* links should be displayed either as a means-end presentation or as an XOR/IOR decomposition presentation. GRL supports both presentations. As *ConcreteGRLspec* is contained by *GRLspec*, the representation choice is global for all GRL diagrams (see Figure 30).

In the absence of a *ConcreteGRLspec*, the default presentation is the XOR/IOR decomposition.

a) *Abstract grammar*

None. This is a concrete syntax metaclass only.

b) *Concrete grammar*

*ConcreteGRLspec* does not have a visual representation, but it impacts the presentation of XOR decomposition links in GRL diagrams (see clause 7.8.7).

#### i) Attributes

- showAsMeansEnd (Boolean): Indicates whether GRL XOR and IOR *Decomposition* links should be displayed with a means-end graphical syntax (true) or simply with an XOR/IOR decomposition graphical syntax (false). Default value is false.

### 7.8.2 GRLGraph

A *GRLGraph* is a container for all actor references, GRL nodes (collapsed actor references and intentional (or indicator) element references) and link references of a GRL diagram. In essence, a GRL graph (or diagram) is a view of the underlying GRL specification (see Figure 30).

a) *Abstract grammar*

None. This is a concrete syntax metaclass only.

b) *Concrete grammar*

*GRLGraph* represents the GRL diagram and as such has no concrete syntax except for *Comments*.

**i) Attributes**

- Inherits attributes from *GRLmodelElement*.

**ii) Relationships**

- Inherits relationships from *GRLmodelElement*.
- Contained by *GRLspec* (1): A *GRLGraph* is contained in the GRL specification.
- Composition of *ActorRef* (0..\*): A *GRLGraph* may contain actor references.
- Composition of *GRLNode* (0..\*): A *GRLGraph* may contain GRL nodes.
- Composition of *LinkRef* (0..\*): A *GRLGraph* may contain link references.
- Composition of *Comment* (0..\*): A *GRLGraph* may contain comments (see Figure 56).

**7.8.3 ActorRef**

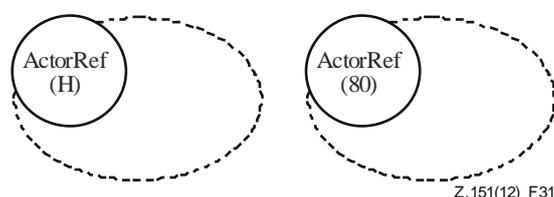
An actor reference (*ActorRef*) shows an actor and its boundary on a GRL diagram (*GRLGraph*). It refers to an *Actor* definition. An actor reference shows the actor's boundary, where intentional elements may be included. In a URN specification, the same actor definition may be referenced many times in the same GRL diagram and in many GRL diagrams (see Figure 30).

*a) Abstract grammar*

None. This is a concrete syntax metaclass only.

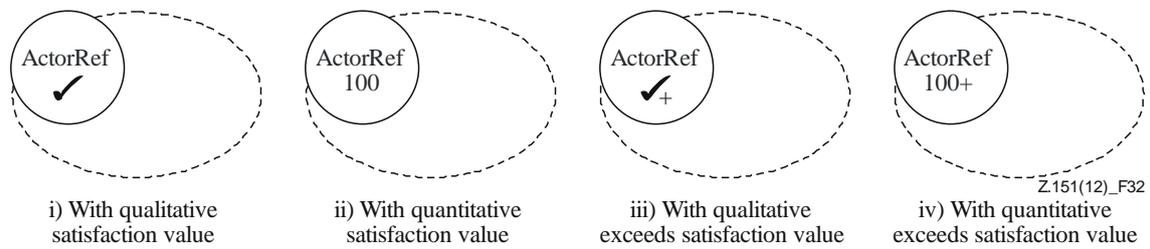
*b) Concrete grammar*

The symbol for an actor reference is a circle, with the name of the actor reference (from superclass *URNmodelElement*) displayed inside the circle, together with its boundary, shown with a dashed-line ellipse (see Figure 31, where the name of the actor reference is ActorRef). The line and fill colours of the actor reference are those of the actor definition's concrete style (*ConcreteStyle*) (from actor definition's superclass *GRLLinkableElement*). If the importanceQualitative information is displayed (from actor definition's superclass *GRLLinkableElement*), then (H) is used for High, (M) for Medium and (L) for Low. None is not displayed. If the quantitative importance information is displayed (from actor definition's superclass *GRLLinkableElement*), then the value is shown in parentheses, but only if greater than zero (see Figure 31).



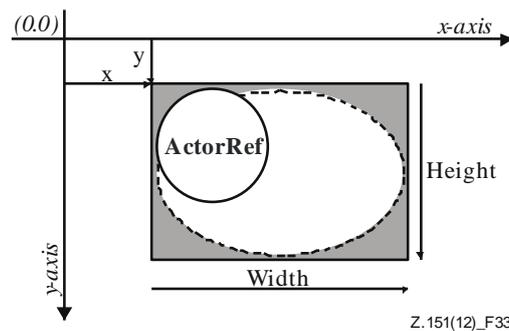
**Figure 31 – Symbol: GRL actor reference with importance values**

When analysing an *EvaluationStrategy* in the GRL specification, the name in the actor reference symbol can be supplemented with a symbol denoting the current satisfaction value of the actor definition. The values to be reported in this way (*qualitativeVal*, *quantitativeVal* and *exceedsVal*) are discussed in clause 11.1. Depending on the nature of the analysis, a qualitative label icon (i) or a quantitative integer value (ii) can be used to annotate the actor reference symbol (see Figure 32). The icons for the qualitative labels are defined in clause 7.5.4 (*QualitativeLabel*). An additional + indicates if the *exceedsVal* is true as shown in (iii) and (iv).



**Figure 32 – Example: GRL actors with satisfaction values**

The coordinate conventions of clause 5.3.2 apply. The top-left corner of the *ActorRef* is indicated by its *Position* ( $x, y$ ) and the bottom-right corner of the actor boundary by its *Position* and *Size* ( $x+width, y+height$ ), as illustrated in Figure 33. The *Label* is not used in this presentation (see clause 7.8.3, numeral d) for label usage). The same layout principles apply also to *IntentionalElementRefs* and *CollapsedActorRefs*.



**Figure 33 – Layout: Position and size of ActorRef, IntentionalElementRef and CollapsedActorRef**

**i) Attributes**

- Inherits attributes from *GRLmodelElement*.

**ii) Relationships**

- Inherits relationships from *GRLmodelElement*.
- Contained by *GRLGraph* (1): An *ActorRef* is contained in one GRL graph.
- Composition of *Position* (1): An *ActorRef* has one position.
- Composition of *Size* (1): An *ActorRef* has one size (for the actor boundary).
- Composition of *Label* (1): An *ActorRef* has one label (see Figure 52).
- Association with *Actor* (1): An *ActorRef* refers to one actor definition.
- Association with *GRLNode* (0..\*): An *ActorRef* may include GRL nodes.

**iii) Constraints**

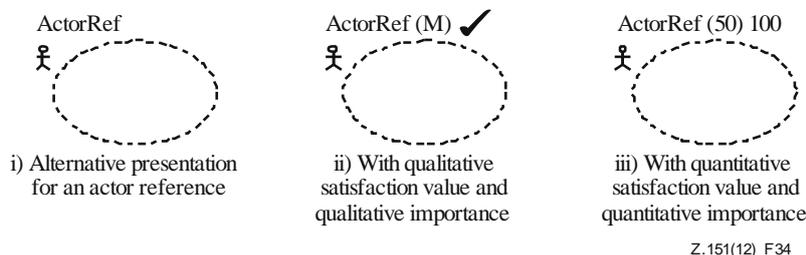
- Inherits constraints from *GRLmodelElement*.
- The name of an *ActorRef* is the same as the name of its referenced *Actor* definition.
- The name of an *ActorRef* including its annotations is inside the actor symbol.
- Rectangles containing the actor symbol and the actor boundary symbol share the same top-left corner.
- The boundary of an *ActorRef* shall not overlap with the boundary of another *ActorRef*.
- The nodes of an *ActorRef* do not include nodes of type *CollapsedActorRef*.

**c) Semantics**

None.

d) *Model*

An alternate way of displaying an *ActorRef*, illustrated in Figure 34, is to omit the actor symbol, to add a stickman icon on the top-left side of the dashed ellipse and to add a *Label* containing the name of the actor reference (from superclass *URNmodelElement*) (i). This label can also contain the qualitative (ii) or quantitative (iii) satisfaction value of the corresponding actor definition (from its superclass *GRLLinkableElement*) resulting from the analysis of an *EvaluationStrategy*. The importance values are again shown in parentheses.



**Figure 34 – Symbol: Alternative presentation for actor references**

The coordinate conventions of clause 5.3.2 apply. The top-left corner of the *ActorRef* is indicated by its *Position* (x, y) and the bottom-right corner by its *Position* and *Size* (x+width, y+height). The bottom-left corner of the *Label* is relative to the *Position* (x-deltaX, y-deltaY) (see Figure 88) for an illustration of these layout principles.

#### 7.8.4 GRLNode

*GRLNode* is an abstraction of intentional element references, indicator references and collapsed actor references in a GRL diagram. GRL nodes except collapsed actor references can be included in actor references and they have a position and a size (see Figure 30).

a) *Abstract grammar*

None. This is a concrete syntax metaclass only.

b) *Concrete grammar*

The concrete syntax for *GRLNode* is further defined in its subclasses.

**i) Attributes**

– Inherits attributes from *GRLmodelElement*.

**ii) Relationships**

– Inherits relationships from *GRLmodelElement*.

– Contained by *GRLGraph* (1): A *GRLNode* is contained by one GRL graph.

– Composition of *Position* (1): A *GRLNode* has one position.

– Composition of *Size* (1): A *GRLNode* has one size.

– Association with *ActorRef* (0..1): A *GRLNode* may be included in one actor reference.

– Association with *LinkRef* (succ, 0..\*): A *GRLNode* may be the source of link references in a diagram.

– Association with *LinkRef* (pred, 0..\*): A *GRLNode* may be the target of link references in a diagram.

– *GRLNode* is a superclass of *IntentionalElementRef* and *CollapsedActorRef*.

**iii) Constraints**

– Inherits constraints from *GRLmodelElement*.

- All instances of *GRLNode* shall appear in one of its subclasses (that is, metaclass *GRLNode* is abstract).
- The *GRLGraph* that contains the *GRLNode* shall be the *GRLGraph* that contains *LinkRefs* associated as *pred*.
- The *GRLGraph* that contains the *GRLNode* shall be the *GRLGraph* that contains *LinkRefs* associated as *succ*.
- If the *GRLNode* is included in one *ActorRef*, then the *GRLGraph* that contains this GRL node shall be the *GRLGraph* that contains this actor reference.
- If the *GRLNode* is included in one *ActorRef*, then the position and size of this GRL node shall be such that the node is entirely contained inside the boundary of the actor reference.

### 7.8.5 IntentionalElementRef

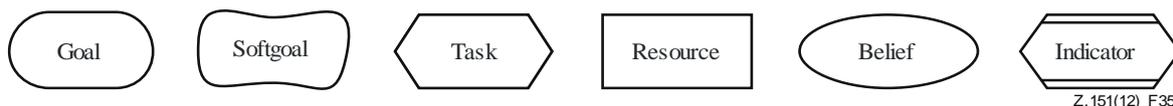
An intentional element reference (*IntentionalElementRef*) shows an intentional element or indicator on a GRL diagram. Its presentation depends on the type of the intentional element it refers to and whether it refers to an indicator. In a URN specification, the same intentional element or indicator definition may be referenced many times in the same GRL diagram and in many GRL diagrams (see Figure 30).

a) *Abstract grammar*

None. This is a concrete syntax metaclass only.

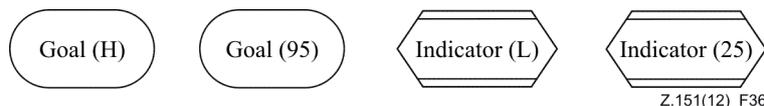
b) *Concrete grammar*

Figure 35 lists the symbols used for indicator references (hexagon with two additional horizontal lines) and intentional element references, which depend on the type of the intentional element definition they refer to (rounded-corner rectangle for Goal, cloud for Softgoal, hexagon for Task, rectangle for Resource and ellipse for Belief).



**Figure 35 – Symbol: GRL intentional element and indicator references**

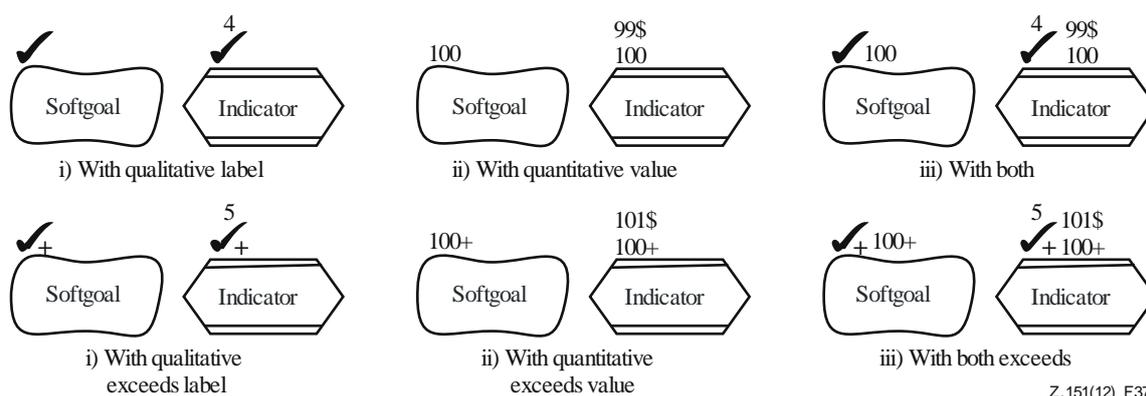
The intentional element or indicator reference name (from superclass *URNmodelElement*) is displayed inside the symbol (in Figure 35, the name of *Indicator* class or the type of each intentional element is used as its name, for illustration purposes). If the intentional element or indicator is contained in an actor, then the importance information may also be displayed following the name. If the importanceQualitative information is displayed (from the superclass *GRLLinkableElement*), then (H) is used for High, (M) for Medium and (L) for Low. None is not displayed. If the quantitative importance information is displayed (from the superclass *GRLLinkableElement*), then the value is shown in parentheses, but only if greater than zero (see Figure 36).



**Figure 36 – Example: GRL intentional elements and indicators with importance values**

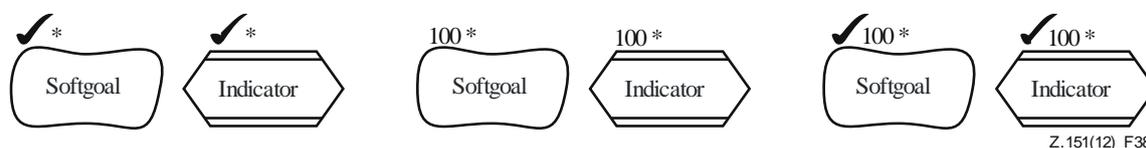
The line and fill colours of the intentional element or indicator reference are those of the intentional element definition's or indicator's concrete style (*ConcreteStyle*) (from the superclass *GRLLinkableElement*).

When analysing an *EvaluationStrategy* in the GRL specification, the intentional element or indicator reference symbol can be supplemented with a symbol denoting the current evaluation value of the referenced intentional element or indicator definition (see Figure 37). The values to be reported in this way (qualitativeVal, quantitativeVal and exceedsVal) are discussed in clause 11.1. Depending on the nature of the analysis, a qualitative label icon (i), a quantitative integer value (ii) or both (iii) can be used to annotate the intentional element or indicator reference symbol. The icons for the qualitative labels are defined in clause 7.5.4 (*QualitativeLabel*). An additional + indicates if the exceedsVal is true as shown in (iv), (v), and (vi). Furthermore for an indicator, the real-world value (from *IndicatorEvaluation*) including its unit (from *Indicator*) or the real-world label (from *IndicatorEvaluation*) is shown above the other symbols depending on the nature of the analysis. The unit for the real-world label is omitted as it typically states something like "5-point Likert scale". Figure 37 depicts a qualitative real-world value (i), a quantitative real-world value with its unit (ii), and both (iii). Note that a quantitative real-world value may be shown with a qualitative evaluation value, for example, if the analysis uses quantitative real-world values and qualitative evaluation values.



**Figure 37 – Example: GRL intentional elements and indicators with satisfaction values**

If the current *EvaluationStrategy* has an *Evaluation* for the referenced intentional element or indicator, then the current annotation is supplemented with a star (\*), which indicates that this is an initial value for this strategy (see Figure 38).



**Figure 38 – Example: GRL intentional elements and indicators with initial satisfaction values**

The coordinate conventions of clause 5.3.2 apply. The top-left corner of the *IntentionalElementRef* is indicated by its *Position* (x, y) and the bottom-right corner by its *Position* and *Size* (x+width, y+height), as explained in Figure 33. The additional annotations are added at a fixed position above the symbol, starting from the left.

**i) Attributes**

- Inherits attributes from *GRLNode*.

**ii) Relationships**

- Inherits relationships from *GRLNode*.

- Association with *GRLContainableElement* (1): An *IntentionalElementRef* references one intentional element or indicator definition.

### iii) Constraints

- Inherits constraints from *GRLNode*.
- The name of an *IntentionalElementRef* is the same as the name of its *GRLContainableElement*.
- If the *IntentionalElementRef* is included by an *ActorRef*, then the referenced *IntentionalElement* definition shall be included by the referenced *Actor* definition.
- The name of an *IntentionalElementRef* including its importance value is inside the intentional element symbol.
- Intentional element and indicator symbols on the same GRL diagram shall not overlap.

### c) Semantics

None.

### d) Model

NOTE – When evaluating a strategy in the GRL specification, the fill colour of the intentional element or indicator symbol is allowed to be overridden temporarily to provide additional visual feedback about the satisfaction level of the referenced intentional element or indicator definition. For example, the following colour scheme could be used: red for **Denied**, orange for **WeaklyDenied**, yellow for **None**, green-yellow for **WeaklySatisfied**, green for **Satisfied**, blue for **Conflict** and white for **Unknown**.

## 7.8.6 CollapsedActorRef

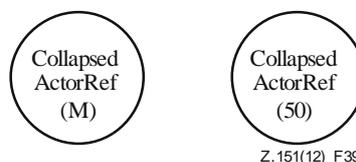
A collapsed actor reference (*CollapsedActorRef*) shows an actor on a GRL diagram. It is presented as a circle, with the actor name displayed inside the circle. A collapsed actor reference in a GRL diagram (*GRLGraph*) refers to an *Actor* definition. Unlike *ActorRef*, a collapsed actor reference does not show the actor's boundary. In a URN specification, the same actor definition may be referenced many times in the same GRL diagram and in many GRL diagrams (see Figure 30).

### a) Abstract grammar

None. This is a concrete syntax metaclass only.

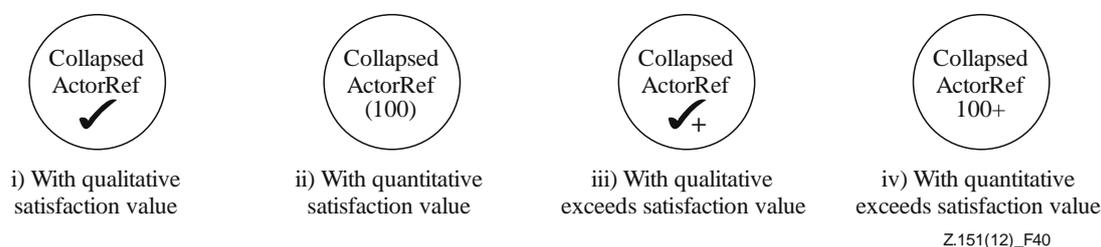
### b) Concrete grammar

The symbol for a collapsed actor reference is a circle, with the collapsed actor reference name (from superclass *URNmodelElement*) displayed inside the circle (see Figure 39, where the name of this reference is *CollapsedActorRef*). The line and fill colours of the collapsed actor reference are those of the actor definition's concrete style (*ConcreteStyle*) (from actor definition's superclass *GRLLinkableElement*). If the importanceQualitative information is displayed (from actor definition's superclass *GRLLinkableElement*), then (H) is used for High, (M) for Medium and (L) for Low. None is not displayed. If the quantitative importance information is displayed (from actor definition's superclass *GRLLinkableElement*), then the value is shown in parentheses, but only if greater than zero.



**Figure 39 – Symbol: GRL collapsed actor reference with importance values**

When analysing an *EvaluationStrategy* in the GRL specification, the name in the actor reference symbol can be supplemented with a symbol denoting the current satisfaction value of the actor definition. The values to be reported in this way (*qualitativeVal*, *quantitativeVal* and *exceedsVal*) are discussed in clause 11.1. Depending on the nature of the analysis, a qualitative label icon (i) or a quantitative integer value (ii) can be used to annotate the actor reference symbol (see Figure 40). The icons for the qualitative labels are defined in clause 7.5.4 (*QualitativeLabel*). An additional + indicates if the *exceedsVal* is true as shown in (iii) and (iv).



**Figure 40 – Example: GRL collapsed actor references with satisfaction values**

The coordinate conventions of clause 5.3.2 apply. The top-left corner of the *CollapsedActorRef* is indicated by its *Position* (x, y) and the bottom-right corner of the actor by its *Position* and *Size* (x+width, y+height), as explained in Figure 33.

#### i) Attributes

- Inherits attributes from *GRLNode*.

#### ii) Relationships

- Inherits relationships from *GRLNode*.
- Association with *Actor* (1): A *CollapsedActorRef* refers to one actor definition.

#### iii) Constraints

- Inherits constraints from *GRLNode*.
- The name of a *CollapsedActorRef* is the same as the name of its referenced *Actor* definition.
- The name of a *CollapsedActorRef* including its annotations is inside the collapsed actor symbol.

### 7.8.7 LinkRef

A link reference (*LinkRef*) displays with a line an element link (*Contribution*, *Dependency* or *Decomposition*) between two GRL linkable elements on a GRL diagram (*GRLGraph*). A link reference is a directed link that connects a source GRL node to a different target GRL node. Link references can be shown as straight lines or as curved lines, and they can contain intermediate bend points. Depending on the nature of the referenced element link, various icons, line ends and labels are displayed (see Figure 30).

#### a) Abstract grammar

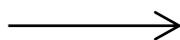
None. This is a concrete syntax metaclass only.

#### b) Concrete grammar

The symbol used to display the *LinkRef* depends on the type of *ElementLink* it represents. In the following definitions, each of the link symbols connects the source symbol (left) to the target symbol (right).

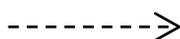
For a *Contribution*, the symbols for contribution and correlation links are different. In both cases, however, the *Label* represents the value of the contribution and/or qualitativeContribution attributes of the *Contribution* (see *ContributionType*, clause 7.4.3).

- If the value of the correlation attribute of the *Contribution* is false, then the symbol is an arrow with the head pointed at the target (see Figure 41).



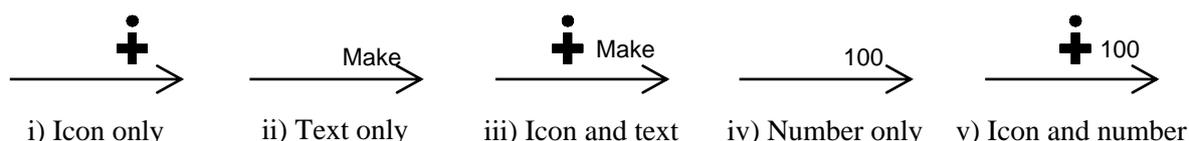
**Figure 41 – Symbol: GRL contribution**

- If the value of the correlation attribute of the *Contribution* is true (i.e., the link is a correlation), then the symbol is a *dashed* arrow with the head pointed at the target (see Figure 42).



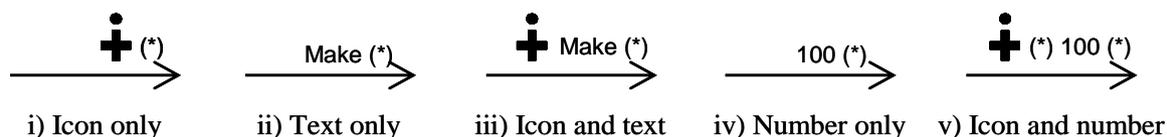
**Figure 42 – Symbol: GRL correlation**

- Depending on the purpose of the GRL model (for quantitative and/or qualitative analysis) the *Label* can include the icon of the qualitativeContribution only (i), a textual representation of the qualitativeContribution only (ii), both the icons and the textual representation of the qualitativeContribution (iii), the numerical contribution value (iv), or both the icon of the qualitativeContribution and the numerical contribution value (v). The choice of presentation should be left to the modeller. This applies to correlations as well. The position of the label is relative to the head of the arrow. The icons for the qualitative labels are defined in clause 7.5.4 (*QualitativeLabel*). A fully satisfied contribution is used here as an example (see Figure 43).



**Figure 43 – Examples: GRL contribution links with contribution values**

- If a *ContributionContext* is selected, the quantitative and qualitative contribution attributes of the *Contributions* affected by the *ContributionContext* are overridden. Overridden contributions are indicated by a \* in parentheses as shown in Figure 44.



**Figure 44 – Examples: GRL contribution links with overridden contribution values**

For a *Dependency*, the symbol is a line with a D on it (see Figure 45). The flat side of the D is pointed at the source. There is no *Label* associated with a dependency link.



**Figure 45 – Symbol: GRL dependency**

For a *Decomposition*, the symbol is a line with a bar crossing it. The *decompositionType* of the target element (i.e., what is being decomposed) is also shown at the end of the line, on the target side. It is shown only once even if there are many decomposition links targeting that element. There is no *Label* associated with a decomposition link.

- If there is no *ConcreteGRLspec* or if the value of the *showAsMeansEnd* attribute of the *ConcreteGRLspec* is false, then the different types of decomposition links are presented as in Figure 46.



**Figure 46 – Symbol: GRL decompositions**

- If the value of the *showAsMeansEnd* attribute of the *ConcreteGRLspec* is true, then XOR and IOR decompositions are shown with an open-headed arrow (i.e., as a *means-end* relationship, see Figure 47). The presentation of the AND decomposition remains unchanged.



**Figure 47 – Symbol: GRL means-end**

The line presentation of a *LinkRef* starts at the source symbol, goes through the ordered list of bend points (*LinkRefBendpoint*) (if any), and stops at the target symbol. The line segments are straight if the value of the *curve* attribute of the link reference is false. If the value of the *curve* attribute of the link reference is true, then the bend points are part of a curved line that connects the start symbol to the target symbol.

#### **i) Attributes**

- Inherits attributes from *GRLmodelElement*.
- *curve* (Boolean): Indicates whether the link should be displayed as a straight line (false) or as a curved line (true). Default value is false.

#### **ii) Relationships**

- Inherits relationships from *GRLmodelElement*.
- Contained by *GRLGraph* (1): A *LinkRef* is contained by one GRL graph.
- Composition of *Label* (0..1): A *LinkRef* may have one label (see Figure 52).
- Composition of *LinkRefBendpoint* (0..\*) {ordered}: A *LinkRef* may have an ordered collection of link reference bend points.
- Association with *ElementLink* (1): A *LinkRef* represents one element link.
- Association with *GRLNode* (source, 1): A *LinkRef* has one source link GRL node.
- Association with *GRLNode* (target, 1): A *LinkRef* has one target link GRL node.

#### **iii) Constraints**

- Inherits constraints from *GRLmodelElement*.
- The source and target GRL nodes shall be different.
- The *LinkRef* has a *Label* if and only if the *ElementLink* to which the *LinkRef* refers is a *Contribution*.
- If the source GRL node is an *IntentionalElementRef*, then the *IntentionalElement* or *Indicator* definition referenced by that source shall be the source of the *ElementLink* to which the *LinkRef* refers.

- If the target GRL node is an *IntentionalElementRef*, then the *IntentionalElement* or *Indicator* definition referenced by that target shall be the destination of the *ElementLink* to which the *LinkRef* refers.
- If the source GRL node is a *CollapsedActorRef*, then the *Actor* definition referenced by that source shall be the source of the *ElementLink* to which the *LinkRef* refers.
- If the target GRL node is a *CollapsedActorRef*, then the *Actor* definition referenced by that target shall be the destination of the *ElementLink* to which the *LinkRef* refers.
- The line connects the border of the source symbol to the border of the target symbol.

c) *Semantics*

None.

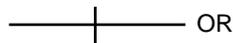
d) *Model*

For dependency links, the D on the line can also be filled (see Figure 48). There is no semantic distinction between a non-filled and a filled D.



**Figure 48 – Symbol: Alternative presentation for GRL dependencies**

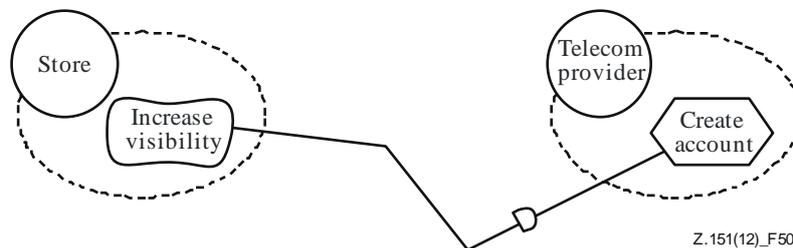
For an IOR decomposition link, the presentation can use OR at the target end instead of IOR, for simplicity (see Figure 49).



**Figure 49 – Symbol: Alternative presentation for IOR decomposition**

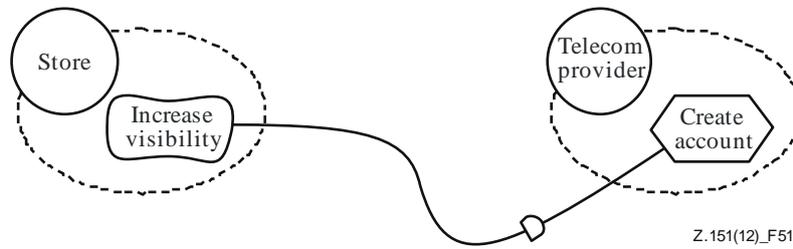
e) *Examples*

Several examples were already presented in clauses 7.4.2, 7.4.4, and 7.4.5. The following GRL diagrams illustrate the effects of bend points on straight and curved lines. This link reference to a dependency that goes from a softgoal to a task has two bend points. If the value of the curve attribute is false, then straight-line segments are used (Figure 50).



**Figure 50 – Example: GRL link with two bend points shown with straight lines**

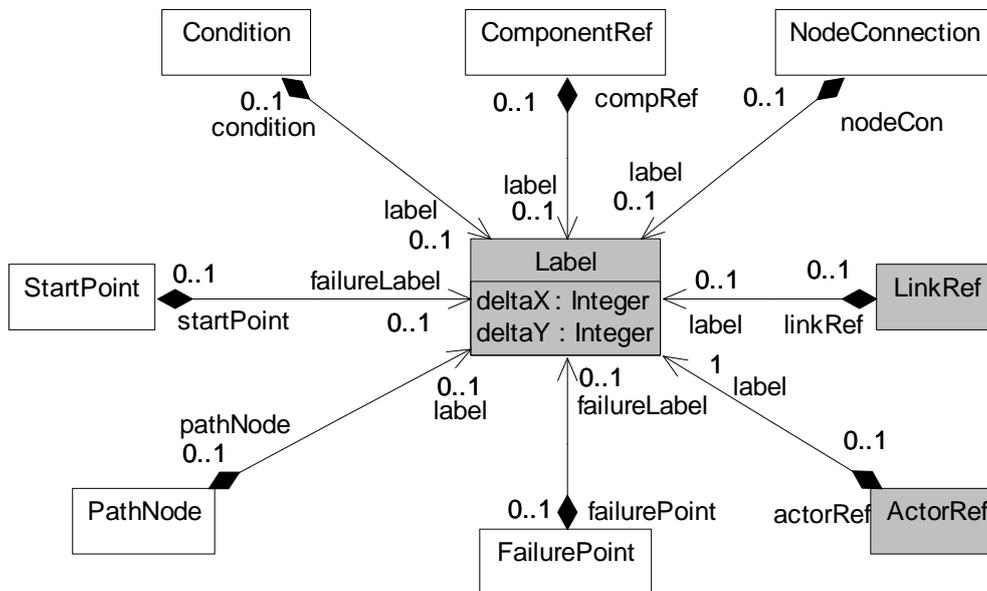
If on the other hand the value of the curve attribute is true, then a curved line that passes through all the bend points is used (Figure 51).



**Figure 51 – Example: GRL link with two bend points shown as a curved line**

### 7.8.8 Label

A *Label* can be attached to many types of URN model elements. A *Label* is contained by an *ActorRef*, *Condition*, *ComponentRef*, *LinkRef*, *NodeConnection* or *PathNode*. An additional label is contained by a *StartPoint* for its list of failures and by a *FailurePoint* for its failure specification. A label indicates the position of the name (or another attribute) of the element relative (in X and Y) to the position of that element, if the element has a position. For elements without position information (i.e., conditions, link references and node connections), the label position is relative to other information (for more details see clauses 6.1.6, 7.8.7 and 8.2.3, respectively) (see Figure 52).



**Figure 52 – Concrete grammar: Label metaclass**

a) *Abstract grammar*

None. This is a concrete syntax metaclass only.

b) *Concrete grammar*

The content displayed by the *Label* depends on the kind of URN model element that contains it and is explained in the respective clauses.

i) **Attributes**

- *deltaX (Integer)*: The relative position, measured in point units, along the horizontal (X) axis of the relevant labelled attribute of the containing URN model element. Can be positive (to the left of the symbol) or negative (to the right of the symbol) (see clause 9.2.2).
- *deltaY (Integer)*: The relative position, measured in point units, along the vertical (Y) axis of the relevant labelled attribute of the containing URN model element. Can be positive (above the symbol) or negative (below the symbol) (see clause 9.2.2).

## ii) Constraints

- A *Label* instance shall be contained in exactly one instance of type *ActorRef*, *Condition*, *ComponentRef*, *LinkRef*, *NodeConnection* or *PathNode*.

### 7.8.9 LinkRefBendpoint

A bend point is a fixed point on a GRL diagram through which a link reference has to pass, therefore breaking the line connecting a source to a target element into several connected segments (straight or curved). A bend point is contained by a *LinkRef* (see Figure 30).

#### a) Abstract grammar

None. This is a concrete syntax metaclass only.

#### b) Concrete grammar

None. However, bend points influence the rendering of link references on a GRL diagram (see clause 7.8.7).

## i) Attributes

- *x* (*Integer*): Horizontal coordinate (on the x-axis) of the bend point, in point units (see clause 9.2.2).
- *y* (*Integer*): Vertical coordinate (on the y-axis) of the bend point, in point units (see clause 9.2.2).

## ii) Constraints

- The line representation of the *LinkRef* that contains the *LinkRefBendpoint* shall pass through the specified bend point.

### 7.8.10 Position

The *Position* metaclass is used to specify the position of various graphical elements in GRL and UCM diagrams. A *Position* is contained by a *PathNode*, *ComponentRef*, *ActorRef* or *GRLNode* (see Figure 53).

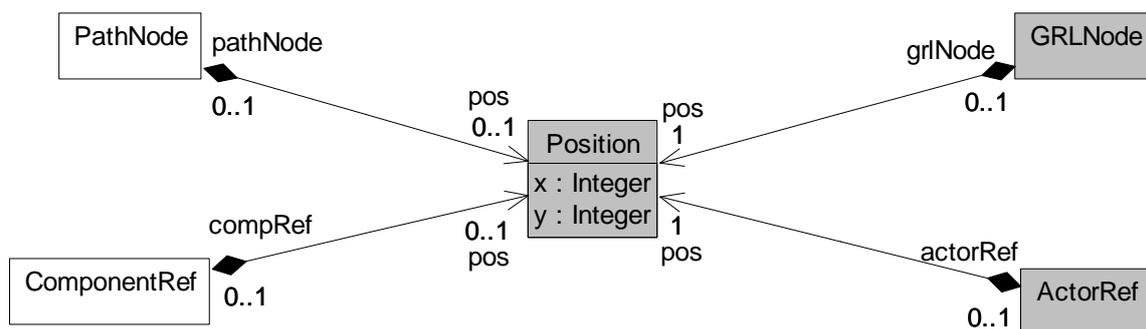


Figure 53 – Concrete grammar: Position metaclass

#### a) Abstract grammar

None. This is a concrete syntax metaclass only.

#### b) Concrete grammar

A *Position* specifies the horizontal and vertical coordinates of the graphical model element where it is contained. These coordinates can be positive or negative. The coordinate conventions of clause 5.3.2 apply.

### i) Attributes

- *x* (*Integer*): Horizontal coordinate (on the x-axis), in point units, of the graphical element containing the *Position* (see clause 9.2.2).
- *y* (*Integer*): Vertical coordinate (on the y-axis), in point units, of the graphical element containing the *Position* (see clause 9.2.2).

### ii) Constraints

- Each *Position* instance is contained in exactly one instance of type *PathNode*, *ComponentRef*, *ActorRef* or *GRLNode*.

## 7.8.11 Size

The *Size* metaclass is used to specify the size of various graphical elements in GRL and UCM diagrams. A *Size* is contained by a *ComponentRef*, *ActorRef* or *GRLNode* (see Figure 54).

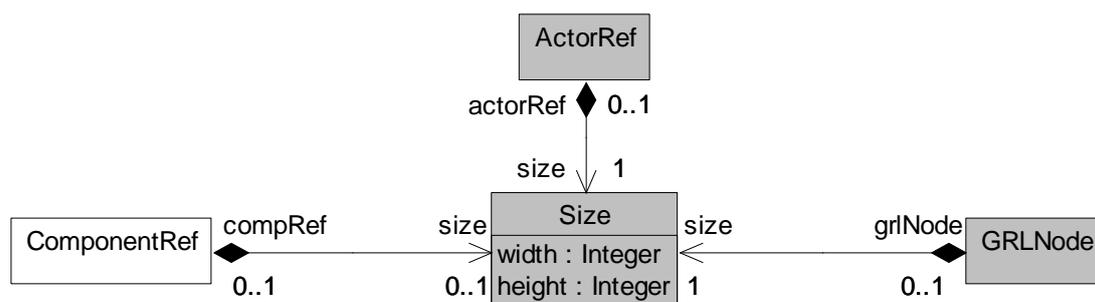


Figure 54 – Concrete grammar: Size metaclass

### a) Abstract grammar

None. This is a concrete syntax metaclass only.

### b) Concrete grammar

A *Size* specifies the width and height of the graphical model element where it is contained. The coordinate conventions of clause 5.3.2 apply.

### i) Attributes

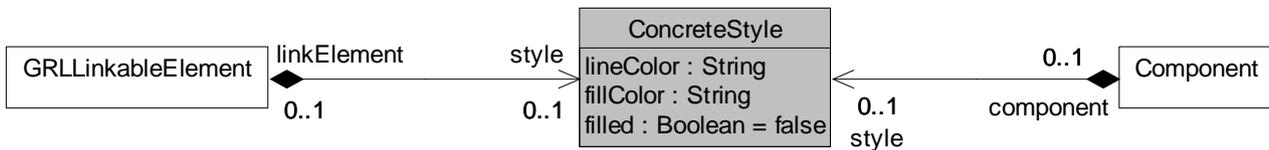
- *width* (*Integer*): The width of the graphical element containing the *Size*, in point units (see clause 9.2.2).
- *height* (*Integer*): The height of the graphical element containing the *Size*, in point units (see clause 9.2.2).

### ii) Constraints

- $width > 0$
- $height > 0$
- Each *Size* instance is contained in exactly one instance of type *ComponentRef*, *ActorRef* or *GRLNode*.

## 7.8.12 ConcreteStyle

The *ConcreteStyle* metaclass is used to specify the colour of various graphical elements in GRL and UCM diagrams. A *ConcreteStyle* is contained by a *GRLLinkableElement* or *Component*. This information is attached to the definitions of intentional elements, indicators, actor definitions and component definitions so their references can be coloured consistently across diagrams (see Figure 55).



**Figure 55 – Concrete grammar: ConcreteStyle metaclass**

a) *Abstract grammar*

None. This is a concrete syntax metaclass only.

b) *Concrete grammar*

A *ConcreteStyle* specifies the fill and line colours of the model element where it is contained, and whether this element should use the fill colour or not. References to a model element can then use these colours across diagrams in a consistent way.

Colours are represented as a string using a red, green and blue colour model. The intensity of each colour component is encoded as a value between 0 (lowest intensity) and 255 (highest intensity), inclusively. The values for red, green and blue in that order are separated by commas. For example, "255,255,0" represents the colour yellow (red=255, green=255, blue=0).

**i) Attributes**

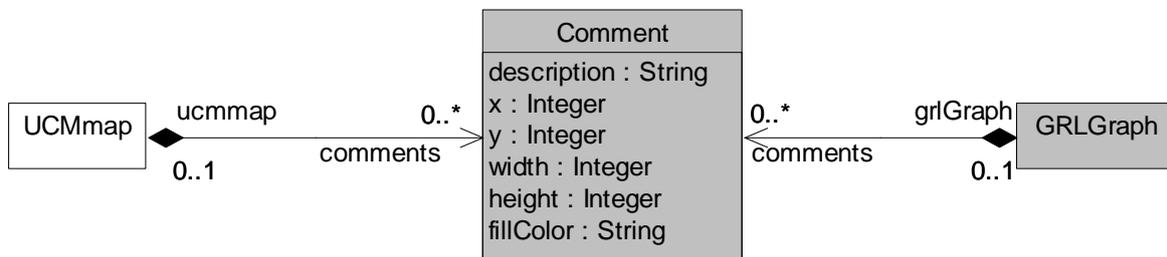
- lineColor (String): Colour of the outside line of the references to the element containing the *ConcreteStyle*.
- fillColor (String): Fill colour of the references to the element containing the *ConcreteStyle*.
- filled (Boolean): Indicates whether the fill colour should be used. Default value is false.

**ii) Constraints**

- Each *ConcreteStyle* instance is contained in exactly one instance of type *GRLLinkableElement* or *Component*.

**7.8.13 Comment**

The *Comment* metaclass is used to add graphical comments to GRL and UCM diagrams. A *Comment* is contained by a *UCMmap* or *GRLGraph* (see Figure 56).



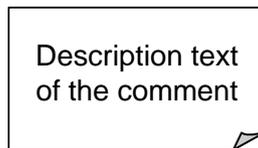
**Figure 56 – Concrete grammar: Comment metaclass**

a) *Abstract grammar*

None. This is a concrete syntax metaclass only.

b) *Concrete grammar*

A *Comment* is illustrated using the following symbol, with the description string displayed in the middle of the symbol (see Figure 57). The description text is wrapped on multiple lines according to the width.



**Figure 57 – Symbol: URN comment**

The coordinate conventions of clause 5.3.2 apply. The top-left corner of the *Comment* is at (x, y) and the bottom-right corner at (x+width, y+height).

**i) Attributes**

- description (String): The text to be displayed in the *Comment*.
- x (*Integer*): Horizontal coordinate (on the x-axis) of the *Comment*, in point units (see clause 9.2.2).
- y (*Integer*): Vertical coordinate (on the y-axis) of the *Comment*, in point units (see clause 9.2.2).
- width (*Integer*): Width of the *Comment*, in point units (see clause 9.2.2).
- height (*Integer*): Height of the *Comment*, in point units (see clause 9.2.2).
- fillColor (String): Fill colour of the *Comment*. The colour conventions of *ConcreteStyle* apply.

**ii) Constraints**

- width > 0
- height > 0
- Each *Comment* instance is contained in exactly one instance of type *UCMmap* or *GRLGraph*.

#### **7.8.14 ConcreteStrategy**

The *ConcreteStrategy* metaclass is used to specify the author of an evaluation strategy (see Figure 30).

*a) Abstract grammar*

None. This is a concrete syntax metaclass only.

*b) Concrete grammar*

There is no graphical visual representation of this metaclass. Optionally, a *ConcreteStrategy* is visualized using the textual syntax for *ConcreteStrategy* as specified in Annex B.

**i) Attributes**

- author (String): The name of the author of the evaluation strategy containing the *ConcreteStrategy*.

## **8 UCM features**

The Use Case Map notation provides a set of URN features that enable the description and analysis of use cases and scenarios. The UCM features are grouped under seven categories:

- UCM basic structural features: Clause 8.1;
- UCM maps and path nodes: Clause 8.2;
- UCM stubs and plug-ins: Clause 8.3;
- UCM components: Clause 8.4;
- UCM scenario definitions: Clause 8.5;

- UCM performance annotations: Clause 8.6;
- UCM concrete grammar metaclasses: Clause 8.7.

NOTE 1 – Many of the concrete grammar metaclasses used by UCM features were already defined for GRL in clause 7.8. Only the ones specific to UCM are defined in clause 8.7.

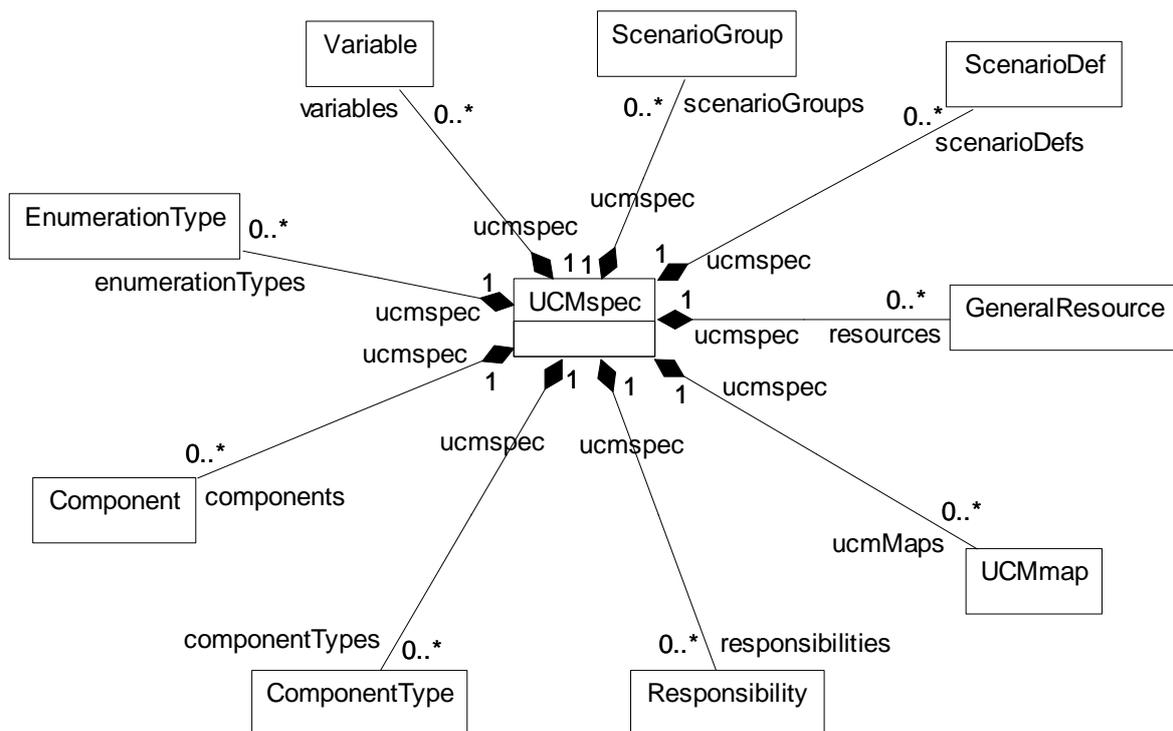
NOTE 2 – In the examples used here to illustrate the UCM concrete syntax, the scenarios flow (are read) from left to right. However, this does not have to be the case with UCM diagrams in general.

## 8.1 UCM basic structural features

The UCM basic structural features describe containers for UCM specifications, as well as definitions of UCM model elements. The abstract syntax metaclasses are presented in this clause. There are no specific concrete grammar metaclasses for these features.

### 8.1.1 UCMSpec

*UCMSpec* serves as a container for the UCM specification elements (see Figure 58).



**Figure 58 – Abstract grammar: UCM specification**

a) *Abstract grammar*

**i) Attributes**

- None.

**ii) Relationships**

- Contained by *URNspec* (1): The *UCMSpec* is contained in the URN specification (see Figure 3).
- Composition of *UCMmap* (0..\*): The *UCMSpec* may contain UCM maps.
- Composition of *Responsibility* (0..\*): The *UCMSpec* may contain responsibility definitions.
- Composition of *ComponentType* (0..\*): The *UCMSpec* may contain component types.

- Composition of *Component* (0..\*): The *UCMSpec* may contain component definitions.
- Composition of *EnumerationType* (0..\*): The *UCMSpec* may contain enumeration types.
- Composition of *Variable* (0..\*): The *UCMSpec* may contain variables.
- Composition of *ScenarioGroup* (0..\*): The *UCMSpec* may contain scenario groups.
- Composition of *ScenarioDef* (0..\*): The *UCMSpec* may contain scenario definitions.
- Composition of *GeneralResource* (0..\*): The *UCMSpec* may contain general resources.

### iii) Constraints

- None.

### b) Concrete grammar

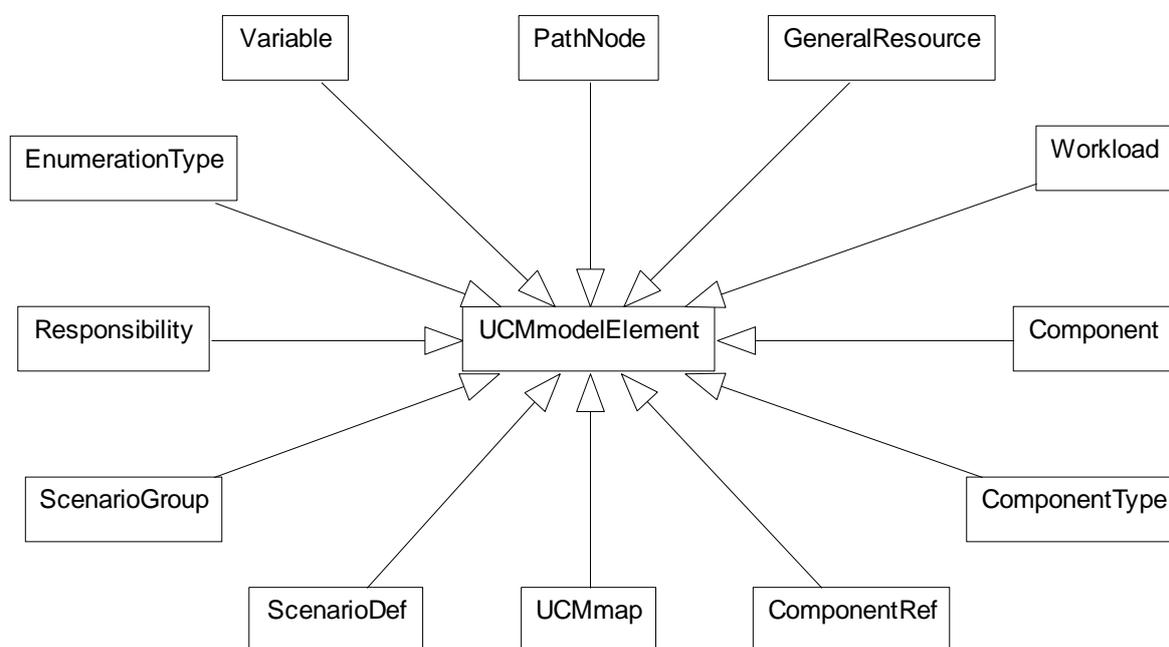
None.

### c) Semantics

None (*UCMSpec* is a structural concept only).

## 8.1.2 UCMmodelElement

A *UCMmodelElement* is a URN model element specialized for UCM concepts (see Figure 59).



**Figure 59 – Abstract grammar: UCM model elements**

### a) Abstract grammar

#### i) Attributes

- Inherits attributes from *URNmodelElement* (see Figure 3).

#### ii) Relationships

- Inherits relationships from *URNmodelElement*.
- *UCMmodelElement* is a superclass of *UCMmap*, *ComponentRef*, *ComponentType*, *Component*, *Workload*, *GeneralResource*, *PathNode*, *Variable*, *EnumerationType*, *Responsibility*, *ScenarioGroup* and *ScenarioDef*.

### iii) Constraints

- Inherits constraints from *URNmodelElement*.
- All instances of *UCMmodelElement* shall appear in one of its subclasses (that is, metaclass *UCMmodelElement* is abstract).

### b) Concrete grammar

The concrete syntax for *UCMmodelElement* is further defined in its subclasses.

### i) Relationships

- Inherits relationships from *URNmodelElement* (see Figure 6).

### c) Semantics

A *UCMmodelElement* is a uniquely identifiable UCM model element that can contain metadata and be linked to other model elements. Its subclasses may have additional attributes and relationships.

## 8.2 UCM maps and path nodes

*UCMmaps* and *PathNodes* enable modelling of scenario behaviour by specifying causal relationships between path nodes on one or more UCM maps. A map contains any number of paths and structural elements (see clause 8.4). Paths express the causal flow of behaviour of a system and may contain several types of path nodes, expressing actions, sequence, alternatives and concurrency as well as the beginning and end of scenarios (see Figure 60).

Hierarchical structuring of maps with the help of stubs and plug-in maps is covered in clause 8.3.

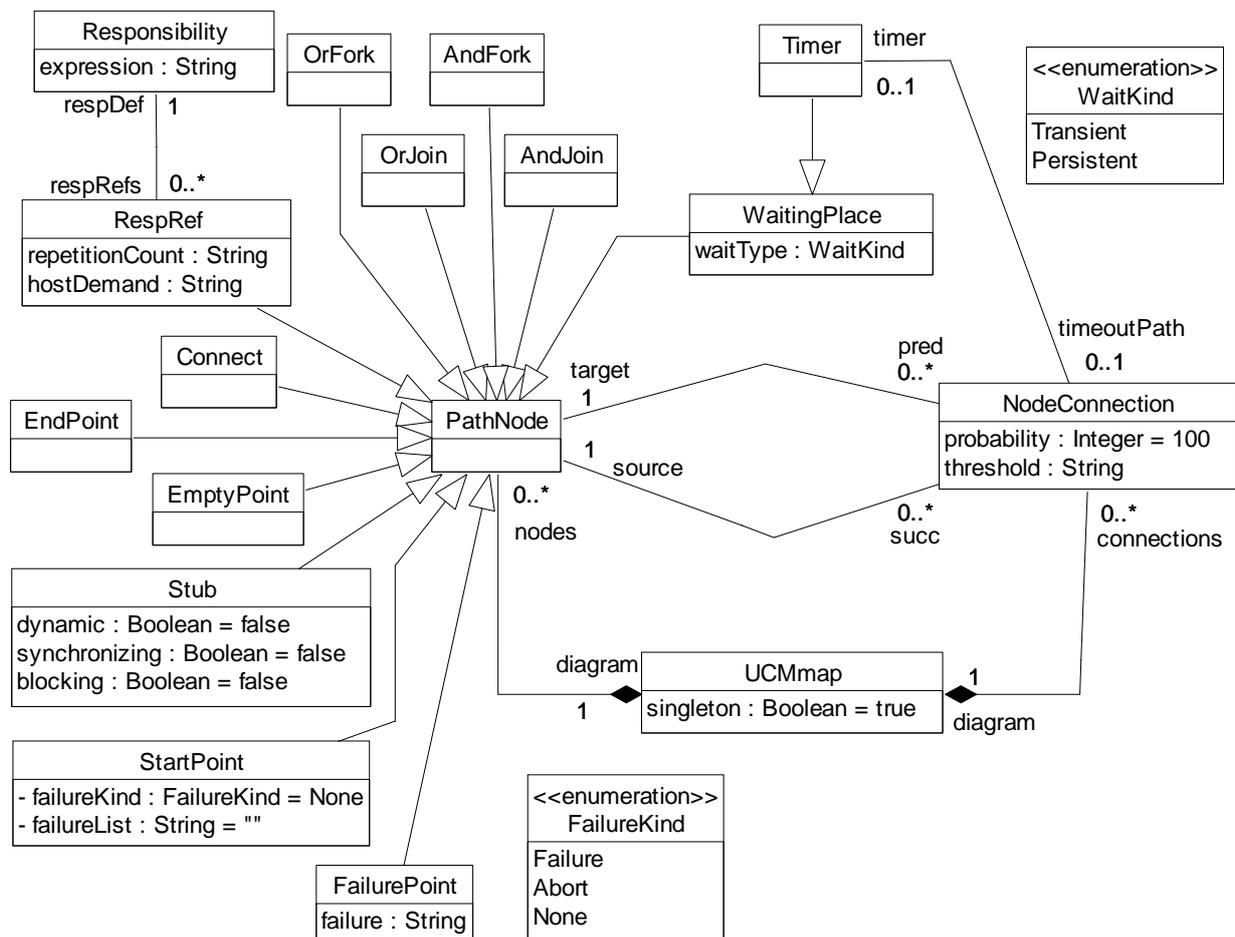


Figure 60 – Abstract grammar: UCM paths and path nodes

## 8.2.1 UCMmap

*UCMmap* serves as a container for all path nodes and component references of a map. A map may be a singleton, i.e., only one runtime instance of it shall exist. A map may be reused as a plug-in map for stubs (see Figures 60, 77 and 85).

### a) *Abstract grammar*

#### i) **Attributes**

- Inherits attributes from *UCMmodelElement* (see Figure 59).
- singleton (Boolean): Indicates whether one (true) or more (false) runtime instances of a *UCMmap* shall exist. Default value is true (i.e., only one runtime instance shall exist).

#### ii) **Relationships**

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *UCMmap* is contained in the UCM specification (see Figure 58).
- Composition of *PathNode* (0..\*): A *UCMmap* may contain path nodes.
- Composition of *NodeConnection* (0..\*): A *UCMmap* may contain node connections.
- Composition of *ComponentRef* (0..\*): A *UCMmap* may contain component references.
- Association with *PluginBinding* (0..\*): A *UCMmap* may be used as a plug-in map in plug-in bindings.

#### iii) **Constraints**

- Inherits constraints from *UCMmodelElement*.

### b) *Concrete grammar*

*UCMmap* represents the map diagram and as such has no concrete syntax except for *Comments* and its name. The name of a *UCMmap* is shown with underline in the top left corner of a map.

#### ii) **Relationships**

- Inherits relationships from *UCMmodelElement*.
- Composition of *Comment* (0..\*): A *UCMmap* may have comments (see Figure 56).

### c) *Semantics*

A *UCMmap* may contain zero or many UCM paths consisting of *NodeConnections* and *PathNodes* which may optionally as well as partially be bound to structural elements called components via *ComponentRefs*. A singleton map exists only once at runtime, i.e., if the same singleton map is used as a plug-in map (see *PluginBinding*, clause 8.3.2) for two different stubs (see *Stub*, clause 8.3.1), the same (and only) runtime instance of the map in the UCM specification is used for both stubs. If, however, the map is not a singleton, a different map runtime instance is used for each different runtime instance of a stub that uses the map as its plug-in map. For a more detailed discussion on runtime instances of maps in the UCM specification, see clause 8.3.1.

### d) *Model*

None.

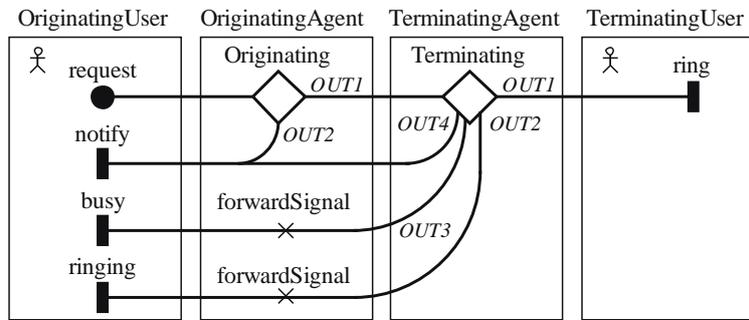
e) *Examples*

The UCM diagrams in Figure 61 present a UCM model of a simple phone system, describing how a phone connection is made between an originating user and a terminating user who have their own phone agents. Each agent handles the features of its associated user. Three features are described on separate plug-in maps:

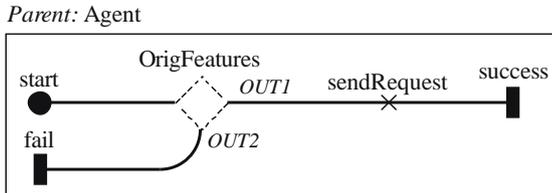
- Originating Call Screening (OCS) is an originating user feature that filters outgoing calls to phone numbers on a screening list (see Figure 61.iv).
- Teen Line (TL) is an originating user feature that filters outgoing calls during a certain time interval (when TL is active) from users who do not have an appropriate personal identification number (PIN) or do not provide the PIN within an appropriate time-frame (see Figure 61.vi).
- Terminating Call Screening (TCS) is a terminating user feature that filters incoming calls from phone numbers on a screening list (see Figure 61.v).

The basic behaviour of the simple phone system is defined on the root map (Simple Connection) of the UCM model (see Figure 61.i) and the root map's two plug-in maps Originating Features and Terminating Features (see Figures 61.ii and 61.iii, respectively). Upon a request from the originating user, the originating agent first executes any Originating Features (OrigFeatures) defined for the user. If no feature exists, the default plug-in is selected (see Figure 61.vii). If a feature fails, the originating user is notified. If none of these features fails or no feature is defined for the user, the originating agent sends a request to the terminating agent.

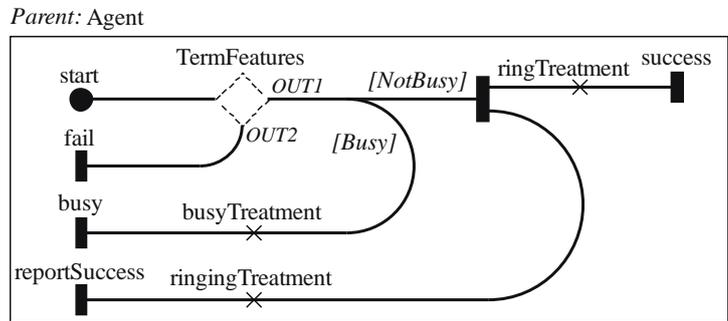
Upon receipt of the request from the originating agent, the terminating agent executes any Terminating Features (TermFeatures) defined for the user. If no feature exists, the default plug-in is selected (see Figure 61.vii). If a feature fails, the originating user is notified. If none of these features fails or no feature is defined for the user, the terminating agent checks whether the terminating user is busy. If the terminating user is busy, the originating user hears the busy signal. If the terminating user is not busy, the phone of the terminating user rings and the originating user hears the ringing signal.



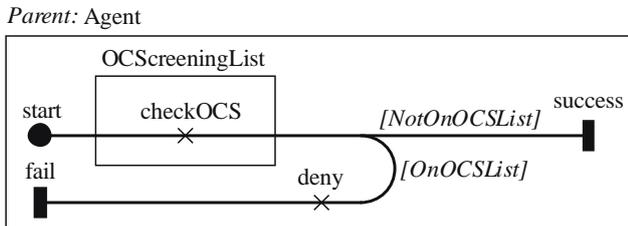
i) Simple connection



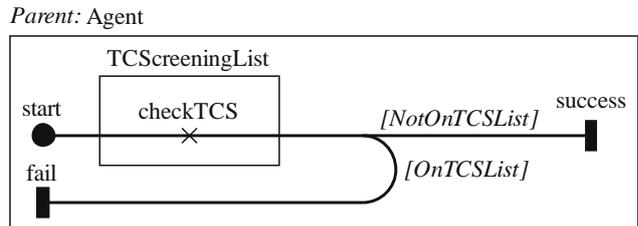
ii) Originating features



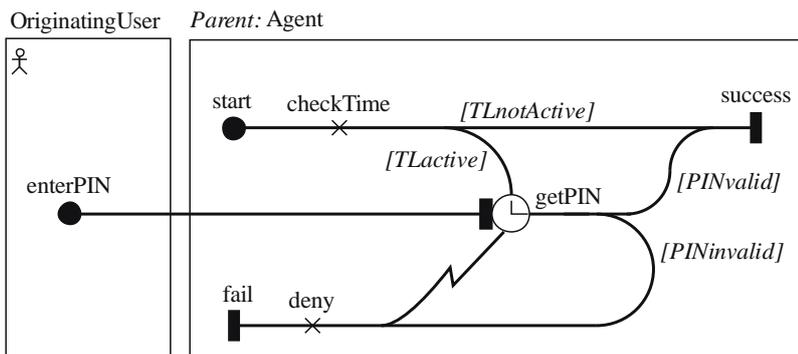
iii) Terminating features



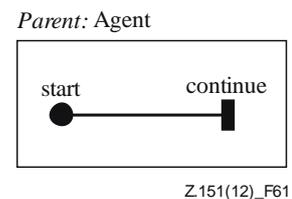
iv) Originating call screening



v) Terminating call screening



vi) Teen line



vii) Default

Figure 61 – Example: UCM model

The plug-in bindings of the UCM model in Figure 61 that connect the in-paths and out-paths of a stub with the start and end points on the plug-in map are defined in Table 4. The component plug-in bindings in this example always connect the stub's component with the "parent:" component on the plug-in map. Furthermore, the following Boolean global variables are specified that more formally define the conditions for the branches of OR-forks and the preconditions of plug-in bindings.

- subOCS: true if the originating user is subscribed to OCS;
- subTL: true if the originating user is subscribed to TL;
- subTCS: true if the terminating user is subscribed to TCS;
- busy: true if the terminating user is busy;

- onOCsList: true if the dialled phone number is on the OCS list;
- onTCSList: true if the originating user's phone number is on the TCS list;
- TLActive: true if TL is active;
- PINValid: true if the PIN is valid.

The OR-fork on the Terminating Features plug-in map uses the variable busy. The OR-fork on the Originating Call Screening plug-in map uses the variable onOCsList. The OR-fork on the Terminating Call Screening plug-in map uses the variable onTCSList. Finally, the OR-forks on the Teen Line plug-in map use the variables TLActive and PINValid.

**Table 4 – Example: Plug-in bindings for the UCM model**

<i>Stub</i>	<i>Plug-in map</i>	<i>In-path/Start point</i>	<i>Out-path/EndPoint</i>	<i>Precondition</i>
Originating	Originating Features	in-path/start	OUT1/success; OUT2/fail	true
Terminating	Terminating Features	in-path/start	OUT1/success; OUT2/reportSuccess; OUT3/busy; OUT4/fail	true
OrigFeatures	Originating Call Screening	in-path/start	OUT1/success; OUT2/fail	subOCS
OrigFeatures	Teen Line	in-path/start	OUT1/success; OUT2/fail	subTL
OrigFeatures	Default	in-path/start	OUT1/continue	not (subOCS or subTL)
TermFeatures	Terminating Call Screening	in-path/start	OUT1/success; OUT2/fail	subTCS
TermFeatures	Default	in-path/start	OUT1/continue	not subTCS

NOTE – The UCM model introduced in this clause is a reasonable, initial model but still needs to be thoroughly tested. This is done with the help of scenario definitions in the example of clause 8.5.2, where the UCM model itself and the global data model of the URN specification (see clause 9.1) are further refined.

### 8.2.2 PathNode

*PathNode* is a UCM model element that represents all possible path nodes on a UCM path. Path nodes may express actions, alternatives (choice points and merge points) and concurrency (parallel branching points and synchronization points) as well as the beginning and end of scenarios. Path nodes may optionally be bound (i.e., allocated) to component references (see Figures 60 and 85).

#### a) Abstract grammar

##### i) Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 59).

##### ii) Relationships

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMmap* (1): A *PathNode* is contained in one UCM map.
- Association with *NodeConnection* (succ, 0..\*): A *PathNode* may be the source of node connections.
- Association with *NodeConnection* (pred, 0..\*): A *PathNode* may be the target of node connections.

- Association with *ComponentRef* (0..1): A *PathNode* may be bound to one component reference.
- *PathNode* is a superclass of *RespRef*, *WaitingPlace*, *FailurePoint*, *OrFork*, *AndFork*, *OrJoin*, *AndJoin*, *Stub*, *EndPoint*, *StartPoint*, *EmptyPoint* and *Connect*.

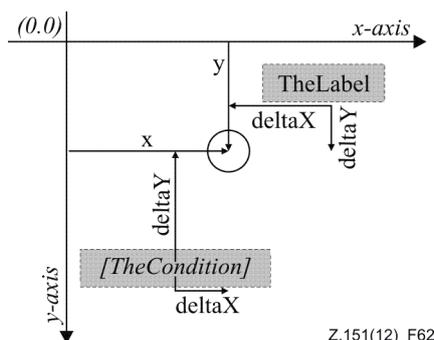
### iii) Constraints

- Inherits constraints from *UCMmodelElement*.
- All instances of *PathNode* shall appear in one of its subclasses (that is, metaclass *PathNode* is abstract).
- The *UCMmap* that contains the *PathNode* shall be the *UCMmap* that contains *NodeConnections* associated as pred.
- The *UCMmap* that contains the *PathNode* shall be the *UCMmap* that contains *NodeConnections* associated as succ.
- If the *PathNode* is included in one *ComponentRef*, then the *UCMmap* that contains this *PathNode* shall be the *UCMmap* that contains the *ComponentRef*.

### b) Concrete grammar

The concrete syntax for *PathNode* is further defined in its subclasses. If a path node has a *Position*, the path node is placed on its *UCMmap* according to *Position* coordinates.

The coordinate conventions of clause 5.3.2 apply. The centre of the *PathNode* is indicated by its *Position* (x, y). The bottom-centre of the *Label* (if any) is relative to the *Position* (x-deltaX, y-deltaY). Similarly, if the *PathNode* also has a *Condition* (*EndPoint* and *StartPoint* sub-classes do), then the bottom-centre of the *Label* of the path node's *Condition* is relative to the path node's *Position* (x-deltaX, y-deltaY). A condition is visualized in italic font and enclosed in square brackets (see Figure 62) for an illustration of these layout principles.



**Figure 62 – Layout: Position, label and condition for PathNode**

### i) Relationships

- Inherits relationships from *UCMmodelElement*.
- Composition of *Label* (0..1): A *PathNode* may have one label (see Figure 52).
- Composition of *Position* (0..1): A *PathNode* may have one position (see Figure 53).
- *PathNode* is a superclass of *DirectionArrow*.

### ii) Constraints

- If the *PathNode* is bound to one *ComponentRef*, then the *Position* of this *PathNode* shall be such that the node is entirely contained inside the boundary of the *ComponentRef*.
- A *PathNode* that is not a *Connect* has exactly one *Position*.

c) *Semantics*

A *PathNode* exists on one UCM map and may optionally be bound to one component reference. Path nodes are arranged in a directed graph with the help of node connections that link source path nodes with target path nodes. Subclasses of path nodes differ among other things in terms of how many path branches (in short branches) may arrive at a path node and how many branches may leave a path node. Further semantics for a path node is therefore defined in the clauses for the subclasses of path nodes.

### 8.2.3 NodeConnection

*NodeConnection* establishes a directed graph of path nodes by linking a source path node with a different target path node. Causality flows along that graph. Node connections have a probability value stating for certain pairs of source and target path nodes the likelihood with which the link between the pair of path nodes is taken in the UCM specification. Node connections also allow the definition of a synchronization threshold for certain pairs of source and target path nodes (see Figures 60 and 77).

a) *Abstract grammar*

**i) Attributes**

- probability (Nat): The probability with which this node connection is taken in the UCM specification. Default value is 100.
- threshold (String): The threshold is an *Integer* expression that indicates the synchronization threshold for an out-path of a synchronizing stub.

**ii) Relationships**

- Contained by *UCMmap* (1): A *NodeConnection* is contained in one UCM map.
- Composition of *Condition* (0..1): A *NodeConnection* may contain one condition.
- Association with *PathNode* (source, 1): A *NodeConnection* has one source path node.
- Association with *PathNode* (target, 1): A *NodeConnection* has one target path node.
- Association with *InBinding* (0..\*): A *NodeConnection* may have in-bindings.
- Association with *OutBinding* (0..\*): A *NodeConnection* may have out-bindings.
- Association with *Timer* (0..1): A *NodeConnection* may represent the timeout path of one timer.

**iii) Constraints**

- probability  $\geq 0$  and probability  $\leq 100$ .
- The value of probability may be less than 100 only for a *NodeConnection* with a source path node of type *OrFork* or *Timer*.
- The threshold shall be empty or an Integer expression, as defined in clause 9.3.
- The threshold shall evaluate to a non-negative Integer value, or it may be empty, in which case it is deemed to evaluate to 0.
- The evaluation value of threshold may be other than 0 only for a *NodeConnection* with a source path node of type *Stub* with its synchronizing attribute equalling to true.
- The source *PathNode* of a *NodeConnection* shall be different from the target *PathNode* of the same *NodeConnection*.
- A *NodeConnection* may have one *Condition* only if the source path node of the *NodeConnection* is of type *OrFork*, *AndFork*, *AndJoin*, *FailurePoint* or *WaitingPlace*.
- A *NodeConnection* may have an *InBinding* only if the target path node of the *NodeConnection* is of type *Stub*.

- A *NodeConnection* may have an *OutBinding* only if the source path node of the *NodeConnection* is of type *Stub*.
- If a *NodeConnection* represents a timeout path of a *Timer*, the source path node of the *NodeConnection* is the same *Timer*.

b) *Concrete grammar*

A *NodeConnection* is rendered as a curved line between the two linked *PathNodes*.

The coordinate conventions of clause 5.3.2 apply. If a *Condition* is defined for the node connection, then the *Label* of the *ConcreteCondition* contained by the condition is displayed in square brackets and italic font. The bottom-centre of the label of that condition is relative to the middle of the curved line linking the source and target path nodes. If no label attribute is defined for the concrete condition, then the square brackets also do not need to be shown.

If a *Label* is defined for the node connection, then the bottom-centre of the label is relative to the middle of the curved line linking the source and target path nodes. The label text is rendered in italic font (see clause 8.3.1).

i) **Relationships**

- Composition of *Label* (0..1): A *NodeConnection* may have one label (see Figure 52).

ii) **Constraints**

- A *NodeConnection* may have one *Label* only if the source path node or the target path node of the *NodeConnection* is of type *Stub*.

c) *Semantics*

The directed graph of *PathNodes* linked by *NodeConnections* is at the core of the traversal of UCMs, as paths in the UCM specification are traversed according to these links and the semantics of the path nodes. In the simplest case, a path node may appear in a node connection once as a source path node and once as a target path node, thus representing the causal ordering of a sequence. Other path nodes may be the source path node or target path node in several node connections, thus representing choice point, merging points, concurrent branching points and synchronization points. Further semantics for a node connection of path nodes is defined in the clauses for the subclasses of *PathNode*.

It is not required that the directed graph is well-nested in terms of its branching and merging constructs, i.e., the path nodes *OrFork*, *OrJoin*, *AndFork* and *AndJoin*. For example, an *OrFork* may never be followed by an *OrJoin*, or an *OrFork* may be followed by an *AndJoin*.

For performance analysis purposes, a node connection may have a probability which expresses the likelihood that the link from the source path node to the target path node is taken. The value of a probability is expressed relative to the probabilities of other node connections with the same source path node. A probability value in per cent is derived by dividing the value of the probability attribute by the sum of the probabilities of all node connections with the same source path node (i.e., 100 means that the link is taken, 0 means that the link is not taken and 75 means that there is a 3:1 chance that the link is taken). Only certain node connections can have probabilities as defined in the Constraints subclause of this clause, i.e., probabilities make sense only for OR-forks and timers. Probabilities have no effect on the regular traversal of UCM models.

The threshold of a node connection is required to specify a part of the synchronizing stub (see clause 8.3.1 for more details).

Node connections may also have a condition which shall be fulfilled (i.e., shall evaluate to true) before the link from the source path node to the target path node can be taken. Only certain node connections can have conditions as defined in the Constraints subclause of this clause, i.e., conditions make sense only for OR-forks, AND-forks, AND-joins, failure points

and waiting places. The conditions for AND-forks and AND-joins are only required to flatten hierarchical UCM specifications (see clause 8.3 for an explanation), and are therefore not considered in the concrete syntax.

Node connections also play a role in the hierarchical structuring of UCM specifications through *InBindings* and *OutBindings*, as explained in clause 8.3.

#### 8.2.4 Responsibility

A *Responsibility* (also referred to as responsibility definition) is a reusable definition of a scenario activity representing something to be performed (operation, action, task, function, etc.) or in other words a step in the scenario. Responsibility definitions are referenced from UCM maps by responsibility references. An expression allows for the formal definition of more detailed behaviour of a responsibility definition with respect to the global data model of the URN specification (see clause 9.1) (see Figure 60).

##### a) *Abstract grammar*

###### i) **Attributes**

- Inherits attributes from *UCMmodelElement* (see Figure 59).
- **expression** (String): The expression of the responsibility definition is described using the URN action language (see clause 9.4). It describes the impact of the responsibility definition on the global data model of the URN specification.

###### ii) **Relationships**

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *Responsibility* definition is contained in the UCM specification (see Figure 58).
- Composition of *Demand* (0..\*): A *Responsibility* definition may contain demands (see Figure 94).
- Association with *RespRef* (0..\*): A *Responsibility* definition may be referenced by responsibility references.

###### iii) **Constraints**

- Inherits constraints from *UCMmodelElement*.
- Any two *Responsibility* definitions cannot share the same name inside a URN specification.
- The name of a *Responsibility* definition cannot be the empty String "".
- The expression shall be an action, as defined in clause 9.4.

##### b) *Concrete grammar*

*Responsibility* definition has no concrete syntax, but responsibility references (see *RespRef*, clause 8.2.5) for the responsibility definition are visualized.

###### i) **Relationships**

- Inherits relationships from *UCMmodelElement*.

##### c) *Semantics*

*Responsibility* definition defines required actions or steps to fulfil a scenario, either informally through its name or more formally with the help of its expression. The actions of the expression may make use of globally defined *Variables*. When the traversal of a scenario path reaches a responsibility reference (*RespRef*), the expression defined in the associated responsibility definition is interpreted. This may change the values of variables in the global

data model of the URN specification. If the expression results in a division by zero, the traversal of the scenario path stops at the responsibility reference and an error is generated.

Responsibility definitions also play a role in the performance analysis of UCM specifications, as their references can make *Demands* on processing resources (see clause 8.6.15).

### 8.2.5 RespRef

*RespRef* is a path node that references a responsibility definition (see Figure 60).

#### a) Abstract grammar

##### i) Attributes

- Inherits attributes from *PathNode*.
- repetitionCount (String): The repetition count is an Integer expression that indicates how often the responsibility reference is repeated at runtime.
- hostDemand (String): The demand is an Integer expression that indicates an average demand on the processing resource of the component reference to which the responsibility reference is bound. The demand is the value of the hostDemand attribute divided by 1000.

##### ii) Relationships

- Inherits relationships from *PathNode*.
- Association with *Responsibility* (1): A *RespRef* has one responsibility definition.

##### iii) Constraints

- Inherits constraints from *PathNode*.
- The name of a *RespRef* shall be the same as the name of its associated *Responsibility* definition.
- The repetitionCount shall be empty or an Integer expression, as defined in clause 9.3.
- The repetitionCount shall evaluate to a positive Integer value or it may be empty, in which case it is deemed to evaluate to 1.
- The hostDemand shall be an Integer expression, as defined in clause 9.3.
- The hostDemand shall evaluate to a non-negative Integer value.
- A *RespRef* is the source *PathNode* of exactly one *NodeConnection*.
- A *RespRef* is the target *PathNode* of exactly one *NodeConnection*.

#### b) Concrete grammar

The symbol for *RespRef* on a UCM path is defined as an **X** with the name of the responsibility reference (from superclass *URNmodelElement*) displayed next to the symbol according to *Label* coordinates (see Figure 63). Optionally, the expression of the *Responsibility* of the *RespRef* is shown next to the name of the responsibility reference.



Figure 63 – Symbol: UCM responsibility reference

##### i) Relationships

- Inherits relationships from *PathNode*.

##### ii) Constraints

- Inherits constraints from *PathNode*.

- A *RespRef* shall have one *Label*.

c) *Semantics*

*RespRef* allows for the reuse of the same *Responsibility* definition in multiple locations on one or more *UCMmaps*.

The *repetitionCount* is an Integer expression that indicates how often the same responsibility reference is repeated. It is equivalent to N consecutive responsibility references to the same responsibility definition placed in a sequence on a UCM path, with N being the resulting value of the repetition count expression.

The *hostDemand* is used for performance analysis and describes the average demand (i.e., the average number of operation requests per traversal of a scenario, divided by 1000) the responsibility reference exerts on a *ProcessingResource*. The demand applies to the processing resource that hosts the component definition that is referenced by the component reference to which the responsibility reference is bound. If the responsibility reference is not bound to a component reference, or if it is bound to a component reference that is not hosted by a processing resource, then the host demand is ignored.

### 8.2.6 StartPoint

*StartPoint* is a path node that denotes the guarded beginning of local scenario behaviour. A start point has a *failureKind* further indicating its semantics and specifies a list of failures. A *StartPoint* may also trigger scenario definitions (see Figures 60, 77 and 92).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *PathNode*.
- *failureKind* (*FailureKind*): The type of start point. Default value is None.
- *failureList* (String): The list of failure variables to which a start point responds. Default value is the empty String "".

**ii) Relationships**

- Inherits relationships from *PathNode*.
- Composition of *Condition* (0..1): A *StartPoint* may contain one precondition.
- Composition of *Workload* (0..1): A *StartPoint* may contain one work load (see Figure 94).
- Association with *InBinding* (0..\*): A *StartPoint* may have in-bindings.

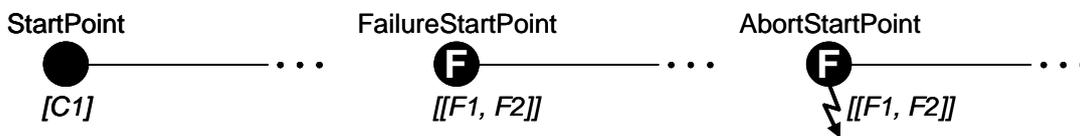
**iii) Constraints**

- Inherits constraints from *PathNode*.
- A *StartPoint* is the source *PathNode* of exactly one *NodeConnection*.
- A *StartPoint* is the target *PathNode* of zero or one *NodeConnection*.
- If a *StartPoint* is the target *PathNode* of one *NodeConnection*, the source *PathNode* of the *NodeConnection* is of type *Connect*.
- If the *failureKind* of a *StartPoint* is Failure or Abort, the *StartPoint* cannot be the target *PathNode* of a *NodeConnection*.
- If the *failureKind* of a *StartPoint* is Failure or Abort, the *StartPoint* cannot have a precondition.
- If the *failureKind* of a *StartPoint* is None, the *failureList* of the *StartPoint* is the empty String "".

- If the failureKind of a *StartPoint* is Failure or Abort, the failureList of the *StartPoint* is a failure list, as defined in clause 9.5. Each failure in the failure list shall be defined as the failure of at least one *FailurePoint*.
- If the failureKind of a *StartPoint* is Failure or Abort, the *StartPoint* does not have any in-bindings.

b) *Concrete grammar*

The symbol for *StartPoint* at the beginning of a UCM path is defined as a filled circle (●) with the name of the start point (from superclass *URNmodelElement*) optionally displayed next to the symbol according to *Label* coordinates of the start point. For a failure start point, the filled circle is overlaid with the letter F, while an abort start point is indicated by the overlaid letter F and an additional lightning bolt. If the start point is not a failure start point or abort start point, the label of the *ConcreteCondition* contained by the *Condition* of the start point (e.g., C1) is also displayed in square brackets and italic font next to the symbol according to *Label* coordinates of the condition (see Figures 64 and 62). If no label attribute is defined for the concrete condition, then the square brackets also do not need to be shown. If the start point is a failure start point or abort start point, the failureList is shown instead of the condition according to the failureLabel coordinates of the start point and enclosed by double square brackets instead of single square brackets. The symbol of a start point is the same as the symbol of a *WaitingPlace*.



**Figure 64 – Symbol: UCM start point, failure start point and abort start point**

i) **Relationships**

- Inherits relationships from *PathNode*.
- Composition of *Label* (failureLabel, 0..1): A *StartPoint* may have one label for its failure list (see Figure 52).

ii) **Constraints**

- Inherits constraints from *PathNode*.

c) *Semantics*

*StartPoint* denotes the beginning of scenario behaviour. The precondition of a start point expresses the conditions for which a scenario is defined. If the precondition is satisfied (true), then the scenario may proceed along the UCM path beginning at the start point. If the precondition is not satisfied (false), then the scenario cannot start.

Start points also denote the beginning of scenarios for failure and exception handling. See *FailureKind* and *FailurePoint* for additional semantics in clauses 8.2.7 and 8.2.17, respectively.

Start points also play a role in: a) the hierarchical structuring of UCM specifications through *InBindings*, as explained in clause 8.3; b) UCM scenario definitions, as explained in clause 8.5; and c) the performance analysis of UCM specifications through *Workloads*, as explained in clause 8.6.

### 8.2.7 FailureKind

A start point can be characterized as Failure, Abort or None (see Figure 60).

a) *Abstract grammar*

i) **Attributes**

- None (enumeration metaclass).

ii) **Relationships**

- Used by *StartPoint*.

iii) **Constraints**

- None.

b) *Concrete grammar*

None (enumeration metaclass). However, it influences the presentation of start points (see clause 8.2.6).

c) *Semantics*

*FailureKind* defines the behaviour of a start point when it is used with a *FailurePoint* for the handling of failures and exceptions. *None* indicates that this is a regular start point as described in clause 8.2.6. *Failure* indicates a failure start point, denoting the beginning of scenario behaviour in response to a failure or exception at a *FailurePoint*. *Abort* indicates an abort start point, i.e., a failure start point that in addition cancels all scenario behaviours in its abort scope. The abort scope is defined as the map of the abort start point as well as all lower level maps as established by the hierarchy of stubs and plug-in maps.

d) *Model*

None.

e) *Examples*

The abort scope depends on the location of the abort start point. The traversal of any path element in the abort scope is stopped when the abort start point is triggered. Assume that all maps in Figure 65 are singleton maps. If the abort start point is specified on Map A, then the abort scope contains Map A and all maps that can be reached from Map A, i.e., Map B, Map C and Map D. In this case, all active traversals in the abort scope are stopped as long as Map A is in the map hierarchy of the traversal. For example, the traversal of the parallel branch with responsibility R1 is stopped as is the traversal of Map D, but only if Map D was reached from Map C, which in turn was reached from Map A. If Map D was reached from Map E, then the traversal is not stopped, because Map A is not in the map hierarchy for that traversal.

If the abort start point is defined on Map B, then only Map B is in the abort scope. If the abort start point is defined on Map C, then Map C and Map D are in the abort scope. If the abort start point is defined on Map D, then only Map D is in the abort scope. If the abort start point is defined on Map E, then Map E and Map D are in the abort scope.

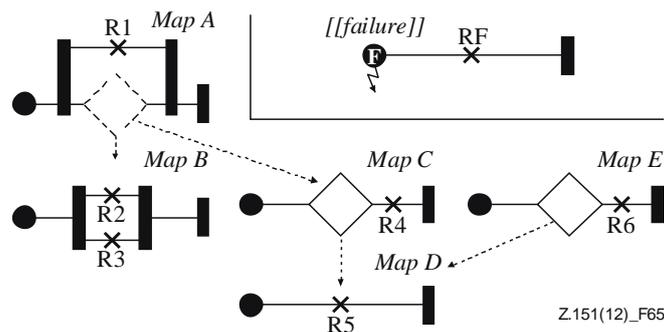


Figure 65 – Example: UCM abort scope

The concept of a visit (see clause 8.3.1) also applies to failure and abort start points on non-singleton maps. When such a failure or abort start point is traversed for the *n*th time, it is the *n*th visit of the map and hence the map instantiated for the *n*th visit is traversed. Consequently, the abort scope then pertains only to those map hierarchies that include the *n*th runtime instance of the map. Any traversal that has a different runtime instance of the map in its current map hierarchy is not affected by the abort start point.

### 8.2.8 EndPoint

*EndPoint* is a path node that denotes the end of local scenario behaviour for which a postcondition may be defined. An *EndPoint* may also be a desired end of a scenario definition (see Figures 60, 77 and 92).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *PathNode*.

**ii) Relationships**

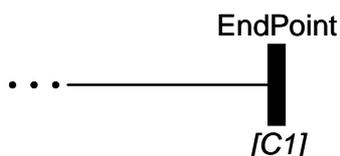
- Inherits relationships from *PathNode*.
- Composition of *Condition* (0..1): An *EndPoint* may contain one postcondition.
- Association with *OutBinding* (0..\*): An *EndPoint* may have out-bindings.

**iii) Constraints**

- Inherits constraints from *PathNode*.
- An *EndPoint* is the source *PathNode* of zero or one *NodeConnection*.
- An *EndPoint* is the target *PathNode* of exactly one *NodeConnection*.
- If an *EndPoint* is the source *PathNode* of one *NodeConnection*, the target *PathNode* of the *NodeConnection* is of type *Connect*.

b) *Concrete grammar*

The symbol for *EndPoint* at the end of a UCM path is defined as a filled bar (I) with the name of the end point (from superclass *URNmodelElement*) optionally displayed next to the symbol according to *Label* coordinates of the end point. The label of the *ConcreteCondition* contained by the *Condition* of the end point (e.g., C1) is also displayed in square brackets and italic font next to the symbol according to *Label* coordinates of the condition (see Figures 66 and 62). If no label attribute is defined for the concrete condition, then the square brackets also do not need to be shown.



**Figure 66 – Symbol: UCM end point**

**i) Relationships**

- Inherits relationships from *PathNode*.

**ii) Constraints**

- Inherits constraints from *PathNode*.

c) *Semantics*

*EndPoint* denotes the end of scenario behaviour. The postcondition of an end point expresses the condition following successful traversal of a given scenario. If the postcondition is satisfied (true), then the scenario was traversed successfully. If the postcondition is not satisfied (false), then the scenario was not traversed successfully.

End points also play a role in: a) the hierarchical structuring of UCM specifications through *OutBindings*, as explained in clause 8.3; and b) UCM scenario definitions, as explained in clause 8.5.

### 8.2.9 OrFork

*OrFork* is a path node that represents a guarded choice point for alternative branches in scenario behaviour (see Figure 60).

a) *Abstract grammar*

i) **Attributes**

– Inherits attributes from *PathNode*.

ii) **Relationships**

– Inherits relationships from *PathNode*.

iii) **Constraints**

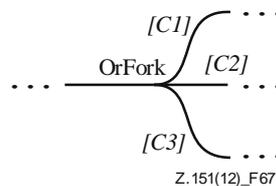
– Inherits constraints from *PathNode*.

– An *OrFork* is the source *PathNode* of one or more *NodeConnections*.

– An *OrFork* is the target *PathNode* of exactly one *NodeConnection*.

b) *Concrete grammar*

The symbol for *OrFork* on a UCM path is defined as a fork with one incoming branch and at least two outgoing branches. The name of the OR-fork (from superclass *URNmodelElement*) is optionally displayed next to the symbol according to *Label* coordinates of the OR-fork. The branch conditions of the OR-fork (e.g., C1, C2, C3) are shown as defined in the concrete syntax of *NodeConnection* (see Figure 67).



**Figure 67 – Symbol: UCM OR-fork**

ii) **Relationships**

– Inherits relationships from *PathNode*.

iii) **Constraints**

– Inherits constraints from *PathNode*.

c) *Semantics*

*OrFork* represents a choice point in the UCM specification with at least two alternative, outgoing branches. Each alternative, outgoing branch (i.e., *NodeConnection*) has a *Condition*. When arriving at the OR-fork during traversal of the UCM path, the conditions are evaluated. If exactly one condition evaluates to true, the alternative branch with that condition is chosen and the traversal continues along that branch. If no condition or more than one condition evaluates to true (non-determinism), then the traversal stops and a warning is generated. If a

condition is not specified for at least one alternative branch (incompleteness), the traversal also stops and an error is generated.

### 8.2.10 OrJoin

*OrJoin* is a path node that represents a merge point for alternative or concurrent branches in scenario behaviour (see Figure 60).

a) *Abstract grammar*

i) **Attributes**

– Inherits attributes from *PathNode*.

ii) **Relationships**

– Inherits relationships from *PathNode*.

iii) **Constraints**

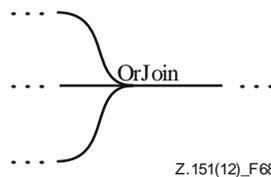
– Inherits constraints from *PathNode*.

– An *OrJoin* is the source *PathNode* of exactly one *NodeConnection*.

– An *OrJoin* is the target *PathNode* of one or more *NodeConnections*.

b) *Concrete grammar*

The symbol for *OrJoin* on a UCM path is defined as a merge of at least two branches into one branch. The name of the OR-join (from superclass *URNmodelElement*) is optionally displayed next to the symbol according to *Label* coordinates (see Figure 68).



**Figure 68 – Symbol: UCM OR-join**

ii) **Relationships**

– Inherits relationships from *PathNode*.

iii) **Constraints**

– Inherits constraints from *PathNode*.

c) *Semantics*

*OrJoin* represents a simple merge point of at least two branches without synchronization. The branches can be either alternative or concurrent branches. When an OR-join is reached during traversal of the UCM path, the traversal simply continues past the OR-join to the next node. If two concurrent branches arrive at an OR-join during the traversal, both will continue and the node past the OR-join will be traversed twice.

### 8.2.11 AndFork

*AndFork* is a path node that represents the beginning of concurrent branches in scenario behaviour (see Figure 60).

a) *Abstract grammar*

i) **Attributes**

– Inherits attributes from *PathNode*.

## ii) Relationships

- Inherits relationships from *PathNode*.

## iii) Constraints

- Inherits constraints from *PathNode*.
- An *AndFork* is the source *PathNode* of one or more *NodeConnections*.
- An *AndFork* is the target *PathNode* of exactly one *NodeConnection*.

### b) Concrete grammar

The symbol for *AndFork* on a UCM path is defined as a filled bar (I) with one incoming branch and at least two outgoing branches. The name of the AND-fork (from superclass *URNmodelElement*) is optionally displayed next to the symbol according to *Label* coordinates (see Figure 69).

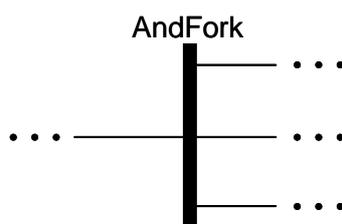


Figure 69 – Symbol: UCM AND-fork

## ii) Relationships

- Inherits relationships from *PathNode*.

## iii) Constraints

- Inherits constraints from *PathNode*.

### c) Semantics

*AndFork* represents a concurrent branching point with at least two concurrent, outgoing branches. When an AND-fork is reached during traversal of the UCM path, the traversal simply continues in parallel past the AND-fork to the next node for each outgoing branch.

## 8.2.12 AndJoin

*AndJoin* is a path node that represents a synchronization point of alternative or concurrent paths in scenario behaviour (see Figure 60).

### a) Abstract grammar

#### i) Attributes

- Inherits attributes from *PathNode*.

#### ii) Relationships

- Inherits relationships from *PathNode*.

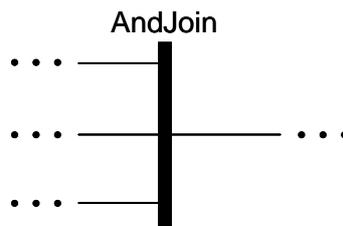
#### iii) Constraints

- Inherits constraints from *PathNode*.
- An *AndJoin* is the source *PathNode* of exactly one *NodeConnection*.
- An *AndJoin* is the target *PathNode* of one or more *NodeConnections*.

### b) Concrete grammar

The symbol for *AndJoin* on a UCM path is defined as a filled bar (I) with at least two incoming branches and one outgoing branch. The name of the AND-join (from superclass

*URNmodelElement*) is optionally displayed next to the symbol according to *Label* coordinates (see Figure 70).



**Figure 70 – Symbol: UCM AND-join**

**i) Relationships**

- Inherits relationships from *PathNode*.

**ii) Constraints**

- Inherits constraints from *PathNode*.

**c) Semantics**

*AndJoin* represents a merge point of at least two incoming branches with synchronization. The incoming branches can be either alternative or concurrent branches. For each incoming branch, the AND-join maintains a counter. The counter for a branch is increased by one when the AND-join is reached along that branch during traversal of the UCM path. Traversal of the UCM path may continue past the AND-join only if the counter for each incoming branch of the AND-join is greater than zero. Before continuing on to the next path node, each counter is decreased by one. The behaviour of an AND-join is best described with the help of counters, but the usage of counters is not mandatory and the same effect may be achieved through other means.

### 8.2.13 EmptyPoint

*EmptyPoint* is a path node that is used to asynchronously connect two paths (see Figure 60).

**a) Abstract grammar**

**i) Attributes**

- Inherits attributes from *PathNode*.

**ii) Relationships**

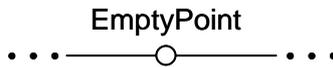
- Inherits relationships from *PathNode*.

**iii) Constraints**

- Inherits constraints from *PathNode*.
- An *EmptyPoint* is the source *PathNode* of one or two *NodeConnections*.
- An *EmptyPoint* is the target *PathNode* of exactly one *NodeConnection*.
- If an *EmptyPoint* is the source *PathNode* of two *NodeConnections*, the target *PathNode* of exactly one of the two *NodeConnections* is of type *Connect*.

**b) Concrete grammar**

The symbol for *EmptyPoint* on a UCM path is defined as a small, empty circle (○). The name of the empty point (from superclass *URNmodelElement*) is optionally displayed next to the symbol according to *Label* coordinates (see Figure 71).



**Figure 71 – Symbol: UCM empty point**

**i) Relationships**

- Inherits relationships from *PathNode*.

**ii) Constraints**

- Inherits constraints from *PathNode*.

**c) Semantics**

*EmptyPoint* does not have any scenario semantics of its own but rather facilitates the asynchronous connection of two paths (see *Connect*). Consequently, the traversal of a UCM path simply passes through an empty point and immediately continues in parallel on to the path nodes following the empty point (i.e., in this case, two *NodeConnections* exist with the empty point as the source node). Furthermore, an empty point bound to a *ComponentRef* does not carry any meaning. If, for example, a path crosses into a component because only an empty point is bound to a *ComponentRef* that references the *Component* definition, it cannot be concluded that the component takes part in the scenario behaviour. The presence of empty points, however, facilitates the layout of complex UCM paths and diagrams.

**d) Model**

An alternative presentation of the *EmptyPoint* is to simply omit the empty point symbol on the path. This makes UCM diagrams less cluttered, without loss of information.

**8.2.14 WaitingPlace**

*WaitingPlace* is a path node that represents a point in scenario behaviour where the continuation of the scenario depends on the fulfilment of a condition or the arrival of a trigger event (i.e., the arrival of a connected UCM path). A waiting place has a *waitType* further indicating its semantics (see Figure 60).

**a) Abstract grammar**

**i) Attributes**

- Inherits attributes from *PathNode*.
- *waitType* (*WaitKind*): The type of waiting place.

**ii) Relationships**

- Inherits relationships from *PathNode*.
- Uses *WaitKind* enumeration.
- *WaitingPlace* is a superclass of *Timer*.

**iii) Constraints**

- Inherits constraints from *PathNode*.
- A *WaitingPlace* that is not a *Timer* is the source *PathNode* of exactly one *NodeConnection*.
- A *WaitingPlace* is the target *PathNode* of one or two *NodeConnections*.
- If a *WaitingPlace* is the target *PathNode* of two *NodeConnections*, the source *PathNode* of exactly one of the two *NodeConnections* is of type *Connect*.

b) *Concrete grammar*

The symbol for *WaitingPlace* on a UCM path is defined as a filled circle (●) with the name of the waiting place (from superclass *URNmodelElement*) optionally displayed next to the symbol according to *Label* coordinates (see Figure 72). The condition of the waiting place (e.g., C1) is shown as defined in the concrete syntax of *NodeConnection*. The symbol of a waiting place is the same as the symbol of a *StartPoint*. Waiting places are visualized the same way regardless of the value of *waitType*. Optionally, the *waitType* is shown in parentheses next to the name of the *WaitingPlace*. For the concrete syntax of the *Timer* subclass of *WaitingPlace*, see clause 8.2.15. See also *Connect*, clause 8.2.18, for further visualizations of waiting places.



**Figure 72 – Symbol: UCM waiting place**

**i) Relationships**

- Inherits relationships from *PathNode*.

**ii) Constraints**

- Inherits constraints from *PathNode*.

c) *Semantics*

*WaitingPlace* represents a location on a UCM path where the traversal of the path stops until a condition is satisfied or a trigger event arrives. The arrival of a trigger event is modelled with a second UCM path that is connected to the waiting place (see *Connect*, clause 8.2.18). A trigger counter keeps track of the arrivals (see *WaitKind* for details, clause 8.2.16). The condition of the waiting place is the *Condition* of the *NodeConnection* with the waiting place as its source.

Upon arrival at a waiting place via the waiting path (for a definition see *Connect*, clause 8.2.18), the waiting counter of the waiting place is increased by one. The initial value of the waiting counter is zero.

The traversal of the waiting path is allowed to continue past the waiting place, if a) the condition evaluates to true or b) both the waiting counter and the trigger counter are greater than zero. When continuing past the waiting place, the waiting counter and the trigger counter are decreased by one. If any counter is already zero, it is not decreased any further. If the condition evaluates to false and the trigger event never arrives, the traversal of the UCM path stops and a warning is generated.

Table 5 gives an overview of the decision process for continuing past the waiting place (PWP) or generating a warning (WAR) based on the condition of the waiting place (CWP) and the trigger counter (TC). The overview assumes that the traversal of the UCM path has already reached the waiting place via the waiting path (i.e., waiting counter > 0). Table 5 is read row by row. For example, the last row says that if the CWP evaluates to false (first column), then a warning is generated if TC equals zero (second column) and traversal continues past the waiting place if TC is greater than zero (third column).

**Table 5 – Overview of waiting place semantics**

<i>CWP</i>	<i>TC = 0</i>	<i>TC &gt; 0</i>
True	PWP	PWP
False	WAR	PWP

The behaviour of a waiting place is best described with the help of counters, but the usage of counters is not mandatory and the same effect may be achieved through other means.

### 8.2.15 Timer

*Timer* is a specialization of the *WaitingPlace* path node where the continuation of the scenario depends on the fulfilment of conditions, the arrival of a trigger event (i.e., the arrival of a connected UCM path), or the occurrence of a timeout (see Figure 60).

#### a) Abstract grammar

##### i) Attributes

- Inherits attributes from *WaitingPlace*.

##### ii) Relationships

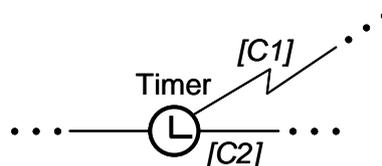
- Inherits relationships from *WaitingPlace*.
- Association with *NodeConnection* (0..1): A *Timer* may have one node connection representing its timeout path.

##### iii) Constraints

- Inherits constraints from *WaitingPlace*.
- A *Timer* is the source *PathNode* of one or two *NodeConnections*.

#### b) Concrete grammar

The symbol for *Timer* on a UCM path is defined as a clock symbol ( $\oplus$ ) with the name of the timer (from superclass *URNmodelElement*) optionally displayed next to the symbol according to *Label* coordinates of the timer. Optionally, the *waitType* is shown in parentheses next to the name of the *Timer*. The branch conditions of the timer (e.g., C1, C2) are shown as defined in the concrete syntax of *NodeConnection*. The optional timeout path of a timer is rendered as a zigzag path. See also *Connect*, clause 8.2.18, for further visualizations of timers.



**Figure 73 – Symbol: UCM timer with timeout path**

##### i) Relationships

- Inherits relationships from *WaitingPlace*.

##### ii) Constraints

- Inherits constraints from *WaitingPlace*.

#### c) Semantics

The semantics of a *Timer* overrides the semantics defined for *WaitingPlace*. A timer represents a location on a UCM path where the traversal of the path stops until conditions are satisfied or a trigger event arrives. The arrival of a trigger event is modelled with a second

UCM path that is connected to the waiting place (see *Connect*, clause 8.2.18). A trigger counter keeps track of the arrivals (see *WaitKind* for details, clause 8.2.16). The conditions of the waiting place are the *Conditions* of the *NodeConnection* with the timer as its source. There is one condition for the timeout path and one condition for the regular path.

Upon arrival at a timer via the waiting path (for a definition see *Connect*, clause 8.2.18), the waiting counter of the timer is increased by one. The initial value of the waiting counter is zero.

The traversal of the waiting path is allowed to continue past the timer along the regular path if, a) the condition of the regular path evaluates to true or b) the condition of the regular path evaluates to false, the condition of the timeout path evaluates to false and both the waiting counter and the trigger counter are greater than zero.

The traversal of the waiting path is allowed to continue past the timer along the timeout path if, a) the condition of the regular path evaluates to false and the condition of the timeout path evaluates to true or b) the condition of the regular path evaluates to false, the condition of the timeout path evaluates to false, the waiting counter is greater than zero and the trigger counter is zero.

Table 6 gives an overview of the decision process for selecting either the regular path (RP) or the timeout path (TOP) for continuation of the traversal of the UCM path based on the condition for the regular path (CRP), the condition for the timeout path (CTOP), and the trigger counter (TC). The overview assumes that the traversal of the UCM path has already reached the timer via the waiting path (i.e., waiting counter > 0). Table 6 is read row by row. For example, the last row says that if the CRP evaluates to false (first column) and the CTOP evaluates to false (second column), then the timeout path is taken if TC equals zero (third column) and the regular path is taken if TC is greater than zero (fourth column).

**Table 6 – Overview of timer semantics**

<i>CRP</i>	<i>CTOP</i>	<i>TC = 0</i>	<i>TC &gt; 0</i>
True	True	RP	RP
True	False	RP	RP
False	True	TOP	TOP
False	False	TOP	RP

When continuing past the timer, the waiting counter and the trigger counter are decreased by one. If any counter is already zero, it is not decreased any further.

The behaviour of a timer is best described with the help of counters, but the usage of counters is not mandatory and the same effect may be achieved through other means.

### 8.2.16 WaitKind

A waiting place can be Transient or Persistent (see Figure 60).

a) *Abstract grammar*

**i) Attributes**

- None (enumeration metaclass).

**ii) Relationships**

- Used by *WaitingPlace*.

**iii) Constraints**

- None.

b) *Concrete grammar*

None (enumeration metaclass). However, it influences the presentation of waiting places (see clause 8.2.14) and timers (see clause 8.2.15).

c) *Semantics*

*WaitKind* defines how a trigger path in the case of a *WaitingPlace* that is not a *Timer* and how a release path in the case of a *Timer* are handled (for a definition of trigger path, release path and waiting path see *Connect*, clause 8.2.18).

If the wait kind is *Transient* or *Persistent*, a trigger counter keeps track of how often a scenario has arrived at the waiting place via the trigger or release path. The initial value of the trigger counter is zero.

For transient waiting places, the trigger counter is set to one upon arrival at the waiting place via the trigger or release path, if the waiting counter for the waiting path of a waiting place is greater than zero (see *WaitingPlace* and *Timer*, clauses 8.2.14 and 8.2.15, respectively). In this case, the trigger counter is never greater than one, thus modelling the arrival of a trigger is only taken into account when the scenario is expecting the trigger (i.e., the scenario is already waiting at the waiting place). Otherwise, the trigger is thrown away. For persistent waiting places, the trigger counter is increased by one upon arrival at the waiting place via the trigger or release path. In this case, all triggers are taken into account.

When continuing past the waiting place, the trigger counter behaves the same way for transient and persistent waiting places (for more details see *WaitingPlace* and *Timer*, clauses 8.2.14 and 8.2.15, respectively).

The behaviour of trigger and release paths is best described with the help of counters, but the usage of counters is not mandatory and the same effect may be achieved through other means.

### 8.2.17 **FailurePoint**

*FailurePoint* is a path node that represents a point in scenario behaviour where the continuation of the scenario depends on the occurrence of a failure or exception (see Figure 60).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *PathNode*.
- *failure* (String): The name of the failure indicating which failure or exception occurred.

**ii) Relationships**

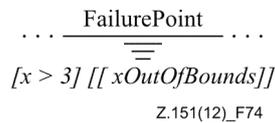
- Inherits relationships from *PathNode*.

**iii) Constraints**

- Inherits constraints from *PathNode*.
- The failure shall be a failure, as defined in clause 9.5.

b) *Concrete grammar*

The symbol for *FailurePoint* on a UCM path is defined as a series of parallel lines with different lengths (similar to the electrical earth symbol) shown just below the path with the name of the failure point (from superclass *URNmodelElement*) optionally displayed next to the symbol according to *Label* coordinates (see Figure 74). The condition of the failure point (e.g.,  $x > 3$ ) is shown as defined in the concrete syntax of *NodeConnection*. The failure of a failure point (e.g., *xOutOfBounds*) is shown similarly to the condition but enclosed by double square brackets and according to the *failureLabel* coordinates of the failure point.



**Figure 74 – Symbol: UCM failure point**

**i) Relationships**

- Inherits relationships from *PathNode*.
- Composition of *Label* (failureLabel, 0..1): A *FailurePoint* may have one label for its failure or exception (see Figure 52).

**ii) Constraints**

- Inherits constraints from *PathNode*.

**c) Semantics**

*FailurePoint* represents a location on a UCM path where the traversal of the path determines whether a failure or exception has occurred by examining the condition of the *FailurePoint*, i.e., the failure condition. The failure condition of the failure point is the negation of the *Condition* of the *NodeConnection* with the failure point as its source. The negation of the failure condition is stored to allow the continuation of the scenario past the failure point if the condition is true (i.e., the failure or exception did not occur) and hence a uniform handling of conditions of node connections by the traversal mechanism.

For example, if the failure condition is  $x > 3$ , i.e., the failure or exception occurred if  $x$  is greater than 3 (see Figure 74), then  $!(x > 3)$  is stored in the condition of the node connection. If this condition now evaluates to true, the scenario is allowed to continue past the failure point because the failure or exception did not occur. This is consistent with all other conditions where continuation of the scenario is also dependent on the condition evaluating to true. If the condition of the node connection evaluates to false, traversal of the scenario stops at the failure point because the failure or exception occurred. The type of failure or exception that occurred is defined by the failure attribute. Hence, if this failure attribute is used in one or more failure or abort start points, the scenario immediately continues in parallel at these failure or abort start points. Consequently, the path that starts at a failure or abort start point describes failure or exception handling. In the case of an abort start point, other scenarios are aborted as described in clause 8.2.7.

Before the scenario is allowed to continue beyond a failure or abort start point during the traversal of a scenario, the map hierarchy of the traversal needs to be taken into account.

- Situation A: If the map of the failure point is the same as the map of the failure or abort start point, then the map hierarchy does not need to be adjusted, i.e., it remains the same as the map hierarchy at the time the traversal reached the failure point. The scenario continues with the same runtime instance of the map.
- Situation B: If the failure or abort start point is on a different map than the failure point, then the map hierarchy needs to be adjusted. In this case, the parent map of the current map is compared against the map of the failure or abort start point. If there is a match, then the scenario is deemed to have exited the current map and returned to the parent map where the scenario continues at the failure or abort start point. The map hierarchy is reduced accordingly. If there is no match, then the next parent map is tried and so on. The traversal continues with the runtime instance of the matched map in the map hierarchy.
- Situation C: If there is no match at all and the map hierarchy has been exhausted, then the failure and abort start point is deemed to be at the root level, the existing map hierarchy is discarded, and the traversal continues on the map of the failure or abort start

point. The map hierarchy hence contains only that map. If this map is a singleton, the traversal continues in a straightforward manner with the only runtime instance available for a singleton map. If this map is a non-singleton, the choice of runtime instance depends on the visit (see clause 8.3.1). When the failure or abort start point is traversed for the  $n^{\text{th}}$  time, then the map instantiated for the  $n^{\text{th}}$  visit is traversed. The traversal mechanism issues a warning in this case.

d) *Model*

None.

e) *Examples*

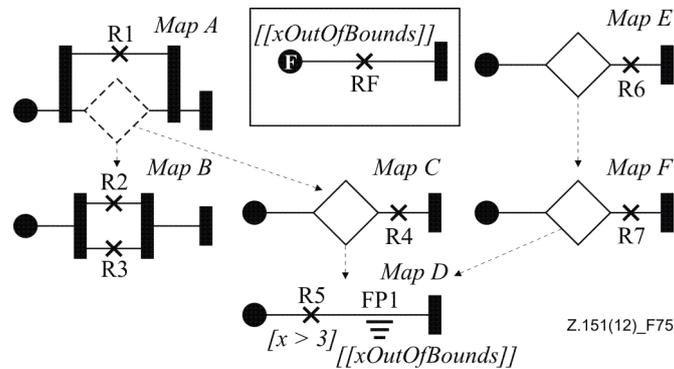
A failure point FP1 is defined on Map D in the example in Figure 75. When the scenario reaches the failure point, the failure condition (i.e.,  $x > 3$ ) is examined. If  $x$  is not greater than 3, no failure or exception occurred and the scenario continues past the failure point. If  $x$  is greater than 3, a failure or exception occurred and the traversal stops at the failure point. What happens next depends on the failure attribute of the failure point. If the failure is `xOutOfBounds`, then the scenario continues at the failure start point shown in Figure 75 because the `failureList` attribute of the failure start point also specifies the same failure. If the failure point specifies a different failure, then the scenario terminates at the failure point with a warning, because the failure or exception is not handled anywhere on the maps shown in Figure 75.

When the scenario continues at the failure start point, the map hierarchy of the traversal needs to be adjusted. Table 7 summarizes various required adjustments of the map hierarchy depending on which map the failure point is specified in the example in Figure 75. For example, if the failure point is specified on Map C, then the map hierarchy is adjusted to "Map A – Map C" if the failure point was reached when traversing "Map A – Map C – Map D". The scenario hence continues on the runtime instance of Map C as specified in the map hierarchy when the failure point was reached. However, if the failure point was reached when traversing "Map E – Map F – Map D", then the map hierarchy is replaced with "Map C", because Map C is outside the existing map hierarchy. In this case, the concept of a visit (see clause 8.3.1) applies. If the failure or abort start point is traversed for the  $n^{\text{th}}$  time, then the runtime instance of Map C instantiated for the  $n^{\text{th}}$  visit is traversed.

As a second example, consider that the failure point is specified on Map B. In this case, the map hierarchy is replaced with "Map B" and the concept of visit applies, because Map B is outside the map hierarchy for both situations covered by the second and third column in Table 7.

**Table 7 – Adjustment of hierarchy of plug-in maps**

<i>Failure point specified on Map</i>	<i>Map hierarchy when failure point is reached</i>	
	<i>Map A – Map C – Map D</i>	<i>Map E – Map F – Map D</i>
A	Map A	Map A (nth visit)
B	Map B (nth visit)	Map B (nth visit)
C	Map A – Map C	Map C (nth visit)
D	Map D	Map D
E	Map E (nth visit)	Map E
F	Map F (nth visit)	Map E – Map F



**Figure 75 – Example: UCM failure point**

### 8.2.18 Connect

*Connect* is a path node that allows exactly two UCM paths to be connected with each other either synchronously (i.e., in sequence by connecting an *EndPoint* to another path) or asynchronously (i.e., in passing by connecting an *EmptyPoint* to another path) (see Figure 60).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *PathNode*.

**ii) Relationships**

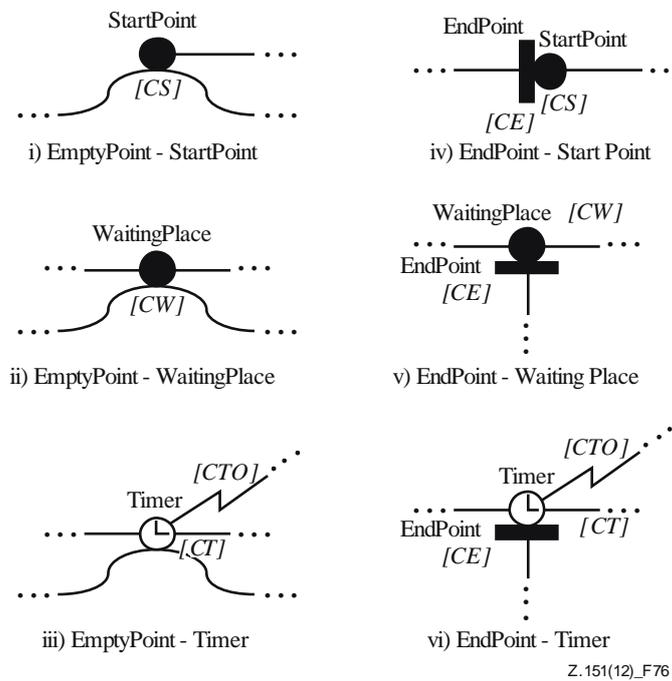
- Inherits relationships from *PathNode*.

**iii) Constraints**

- Inherits constraints from *PathNode*.
- A *Connect* is the source *PathNode* of exactly one *NodeConnection*.
- A *Connect* is the target *PathNode* of exactly one *NodeConnection*.
- If a *Connect* is the source *PathNode* of a *NodeConnection*, the target *PathNode* of the *NodeConnections* is of type *WaitingPlace* or *StartPoint*.
- If a *Connect* is the target *PathNode* of a *NodeConnection*, the source *PathNode* of the *NodeConnections* is of type *EndPoint* or *EmptyPoint*.
- Let P1 be the source *PathNode* of the *NodeConnection* for which a *Connect* is the target *PathNode*. Let P2 be the target *PathNode* of the *NodeConnection* for which the same *Connect* is the source *PathNode*. If at least one of P1, P2 and the *Connect* is bound to a *ComponentRef*, then all three have to be bound to the same *ComponentRef*.

b) *Concrete grammar*

A *Connect* has no concrete syntax as it is not visualized directly. A connect, however, influences the visualization of other path nodes that are linked together by this connect. Figure 76 illustrates all six possible combinations. If an *EmptyPoint* is used to connect two paths together (see examples on left side), the symbol for the empty point is not rendered.



Z.151(12)\_F76

**Figure 76 – Examples: UCM connects**

The second UCM path that either touches a start point or waiting place or that ends with an end point connected to a start point or waiting place in the above examples is called a *trigger path*. The second UCM path that either touches a timer or that ends with an end point connected to a timer in the above examples is called a *release path*. The path segment before a waiting place or timer is called the *waiting path*.

**i) Relationships**

- Inherits relationships from *PathNode*.

**ii) Constraints**

- Inherits constraints from *PathNode*.
- A *Connect* does not have a *Label*.
- A *Connect* does not have a *Position*.
- There shall not be any visual spacing between the symbols for the path nodes before and after the *Connect*.

c) *Semantics*

*Connect* does not have any scenario semantics of its own but rather facilitates the synchronous and asynchronous connection of two paths. Consequently, the traversal of a UCM path simply passes through a connect and immediately continues on to the path node following the connect. An asynchronous connection involves an *EmptyPoint*, while a synchronous connection involves an *EndPoint*. In an asynchronous connection the traversal of the UCM path continues along the trigger or release path regardless of what happens at the connected *StartPoint*, *WaitingPlace* or *Timer*. In a synchronous connection, the traversal of the trigger or release path comes to an end at the end point. Only the traversal of the waiting path continues.

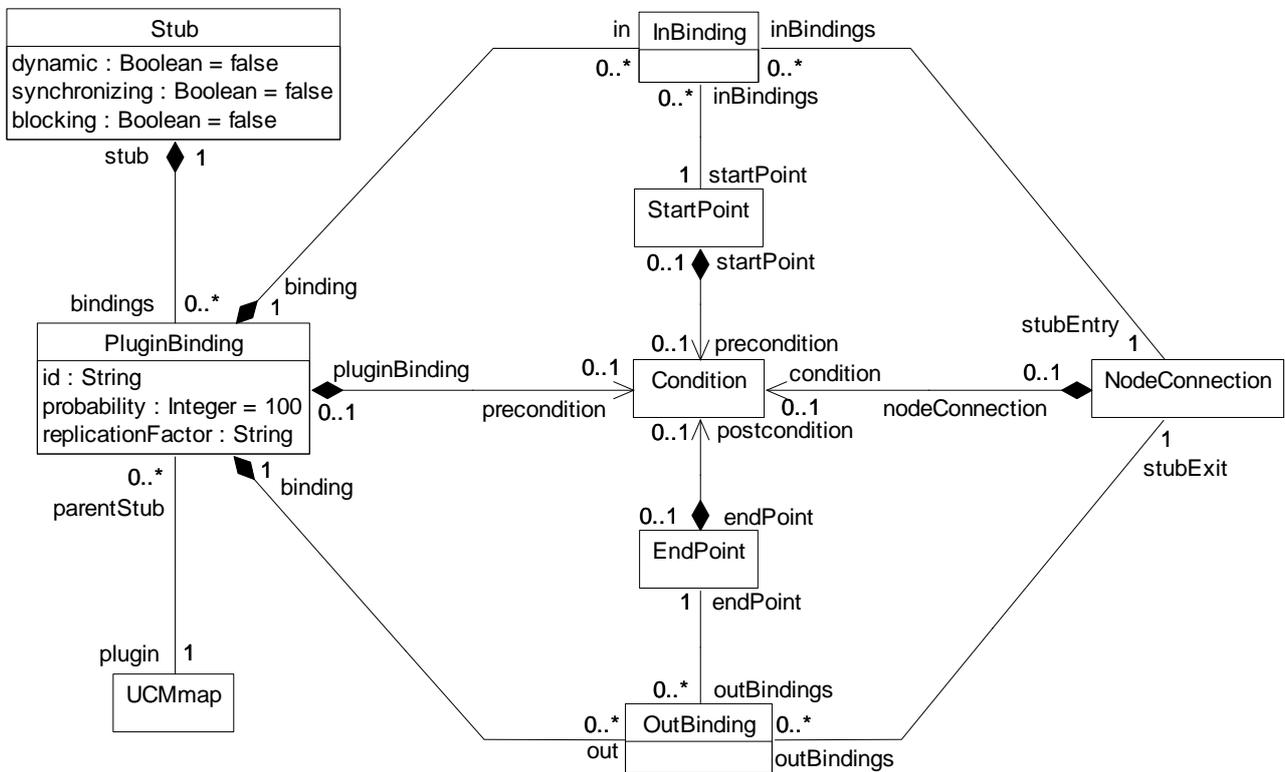
All possible combinations of *NodeConnections* with connects are summarized in Table 8.

**Table 8 – Combinations of node connections with connects**

<i>First node connection</i>		<i>Second node connection</i>		<i>Visualized in</i>
<i>Source node</i>	<i>Target node</i>	<i>Source node</i>	<i>Target node</i>	
Empty Point	Connect	Same Connect	Start Point	Figure 76 (i)
Empty Point	Connect	Same Connect	Waiting Place	Figure 76 (ii)
Empty Point	Connect	Same Connect	Timer	Figure 76 (iii)
End Point	Connect	Same Connect	Start Point	Figure 76 (iv)
End Point	Connect	Same Connect	Waiting Place	Figure 76 (v)
End Point	Connect	Same Connect	Timer	Figure 76 (vi)

### 8.3 UCM stubs and plug-ins

*Stubs* and their *PluginBindings* enable hierarchical structuring of UCM specifications. A *PluginBinding* binds (i.e., connects) model elements on the parent map that contains the stub with models elements on a plug-in map. *PluginBindings* specify *ComponentBindings* (covered in clause 8.4), *InBindings* and *OutBindings*. An *InBinding* binds the in-path of a stub (i.e., a *NodeConnection*) with a start point on the plug-in map, while an *OutBinding* binds the out-path of a stub (another *NodeConnection*) with an end point on the plug-in map (see Figure 77).



**Figure 77 – Abstract grammar: UCM stubs and plug-ins**

#### 8.3.1 Stub

*Stub* is a path node that indicates the presence of hierarchically-structured UCM maps (see Figures 60 and 77).

- a) *Abstract grammar*
  - i) **Attributes**
    - Inherits attributes from *PathNode*.

- dynamic (Boolean): Indicates whether the stub is dynamic (true), i.e., can have more than one plug-in map, or static (false), i.e., can have at the most one plug-in map. Default value is false.
- synchronizing (Boolean): Indicates whether the stub synchronizes its plug-in maps (true) or not (false). Default value is false.
- blocking (Boolean): Indicates whether the stub allows its plug-in maps to be visited more than once at the same time (false) or whether the stub blocks an additional visit (true). Default value is false.

**ii) Relationships**

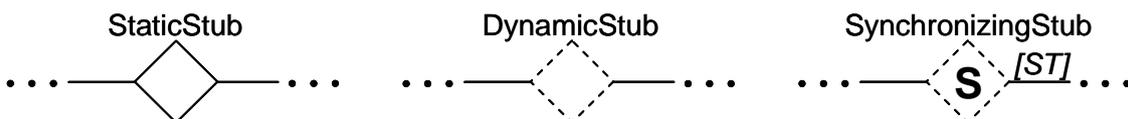
- Inherits relationships from *PathNode*.
- Composition of *PluginBinding* (0..\*): A *Stub* may contain plug-in bindings.

**iii) Constraints**

- Inherits constraints from *PathNode*.
- If synchronizing is true, then dynamic is true.
- If blocking is true, then synchronizing is true.
- If static is true, then the number of *PluginBindings* contained by the *Stub* is zero or one.
- If static is true, then the precondition of *PluginBinding* of the *Stub* is true.
- If static is true, then the replicationFactor of *PluginBinding* of the *Stub* is one.

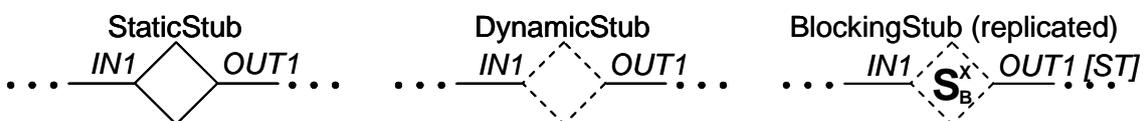
b) *Concrete grammar*

The basic symbol for *Stub* on a UCM path is defined as a diamond symbol ( $\diamond$ ) with the name of the stub (from superclass *URNmodelElement*) optionally displayed next to the symbol according to *Label* coordinates of the stub.



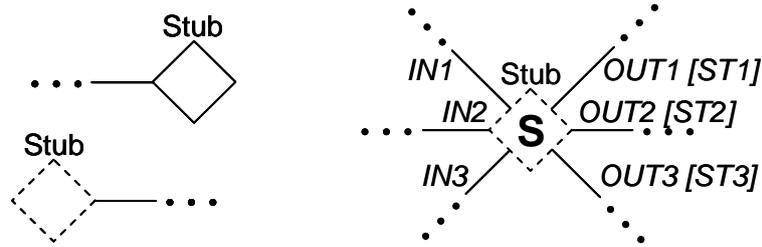
**Figure 78 – Symbol: UCM static, dynamic and synchronizing stubs**

A static stub is rendered with a solid outline, a dynamic stub is rendered with a dashed outline and a synchronizing stub is rendered with the letter S inside the stub symbol (see Figure 78). IN and OUT labels may optionally be shown for all kinds of stubs, but the synchronization threshold shall be shown for an out-path of a synchronizing stub if the threshold has been defined by the modeller (see Figure 79). The synchronization threshold is defined by the threshold attribute of the *NodeConnection* that represents an out-path of the stub (e.g., ST). The synchronization threshold is appended at the end of the OUT label. IN and OUT labels are shown as defined in the concrete syntax of *NodeConnection*. A blocking stub adds a B rendered in subscript to the symbol of the synchronizing stub, while replicated plug-in maps add an X rendered in superscript to the symbol of a dynamic stub. Replicated plug-in maps are defined by the replicationFactor attribute of *PluginBinding*.



**Figure 79 – Symbol: UCM stubs with further annotations**

The incoming node connection of the stub is called *in-path*, while the outgoing node connection is called *out-path*. Any kind of stub may have zero or many in-paths and out-paths as shown in Figure 80.



**Figure 80 – Example: UCM stubs with different numbers of in-paths and out-paths**

**i) Relationships**

- Inherits relationships from *PathNode*.

**ii) Constraints**

- Inherits constraints from *PathNode*.

**c) Semantics**

*Stub* represents hierarchical structuring of UCM specifications through containment of plug-in maps. When the traversal of a UCM path reaches a stub, the traversal continues with the plug-in maps of the stubs. When the traversal reaches an end point on a plug-in map, the traversal may return to the map of the stub (i.e., the parent map) and proceed past the stub. The exact binding of the parent map to the plug-in map is specified with the help of *PluginBindings*, *ComponentBindings* (covered in clause 8.4), *InBindings* and *OutBindings*.

**i) Types of stubs**

Several types of stubs exist as explained in the following paragraphs.

- A static stub has at the most one plug-in map that cannot be replicated and that is always selected (see *PluginBinding*) when the traversal of the UCM path reaches the static stub.
- A dynamic stub may have many plug-in maps that can be replicated and that are selected based on the preconditions of their *PluginBindings* when the traversal of the UCM path reaches the dynamic stub. The selected plug-in maps of the stub are traversed in parallel.
- A synchronizing stub is a dynamic stub that in addition synchronizes its plug-in maps before the traversal of the UCM path is allowed to continue past the stub. By default, a synchronizing stub expects as many plug-in maps as were selected to arrive at an out-path before the scenario is continued (i.e., not necessarily all plug-in maps defined for the stub). A synchronization threshold (the threshold attribute of the *NodeConnections* representing the out-paths of the stub) may override the default. The synchronization threshold is an Integer expression greater than zero and can be defined for each out-path of a stub. The synchronization threshold may be greater than the number of plug-in maps defined for the stub, because a single plug-in map may arrive at a stub's out-path multiple times due to loops. All plug-in maps that arrive at a stub's out-path after its synchronization threshold has been reached are ignored.
- Finally, a blocking stub is a synchronizing stub that does not allow its plug-in maps to be visited more than once at the same time.

A *visit* in the context of synchronizing and blocking stubs is defined by how often an in-path of the stub has been traversed. If an in-path is traversed the first time, then it is the first visit of the stub. If the same in-path is traversed the *n*th time, then it is the *n*th visit of the stub. If another in-path of the stub is traversed for the first time, then it is the first visit of the stub. Plug-in maps that have been instantiated because of a visit are said to belong to the visit.

The concept of a visit also applies to unconnected start points on non-singleton plug-in maps regardless of the type of the plug-in map's stub. When such an unconnected start point is traversed for the *n*th time, it is the *n*th visit of the plug-in map and hence the plug-in map instantiated for the *n*th visit is traversed.

## ii) Plug-in map runtime instances

Plug-in maps that are plugged into a stub are instantiated when the stub is reached the first time during the traversal of a UCM path. The fact that stubs are often used to restructure a complicated map implies that a stub runtime instance shall contain not more than one runtime instance of a plug-in map at any time (with the exception of replicated plug-in maps which require that the specified number of runtime instances is created). This also applies to a stub that is used in a loop. The "not more than one map runtime instance per stub runtime instance" rule ensures the equivalence of a plug-in maps-based UCM specification with its flattened representation that uses only one single map. Synchronizing stubs are an exception for this rule and are discussed later on in this clause.

Since maps can be designated as singleton maps, there are three cases a modeller may want to capture, as illustrated in the example below (see Figure 81).

- Map G is a singleton and therefore the same runtime instance of this map is used by the stubs on Map A, Map B and Map C. The same applies to Map H and the stubs on Map D, Map E and Map F.
- Map I, on the other hand, is not a singleton. Therefore, the stubs on Map G and Map H use different runtime instances of Map I.
- Finally, if a group of stubs are to use the same runtime instance of a plug-in map, this is achieved with an intermediary layer of singleton maps. For example, the group of stubs on Map A, Map B and Map C uses the same runtime instance of Map I but the group of stubs on Map D, Map E and Map F uses a different runtime instance of Map I.

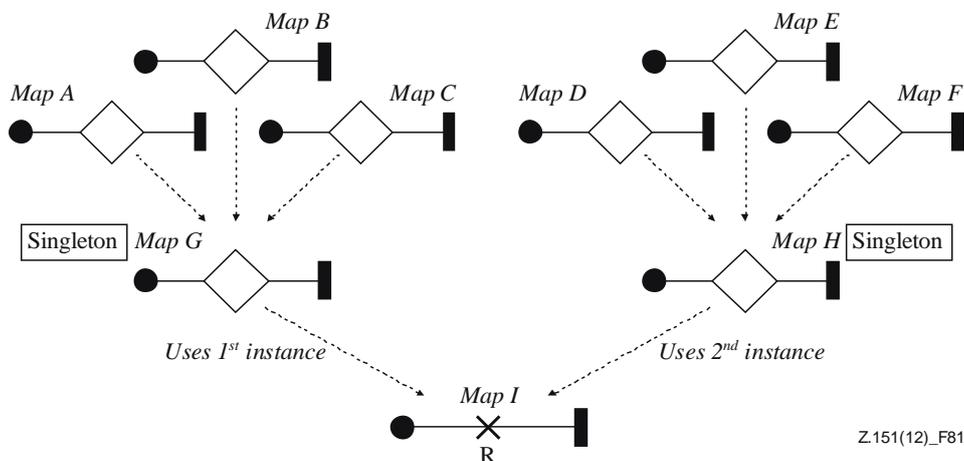
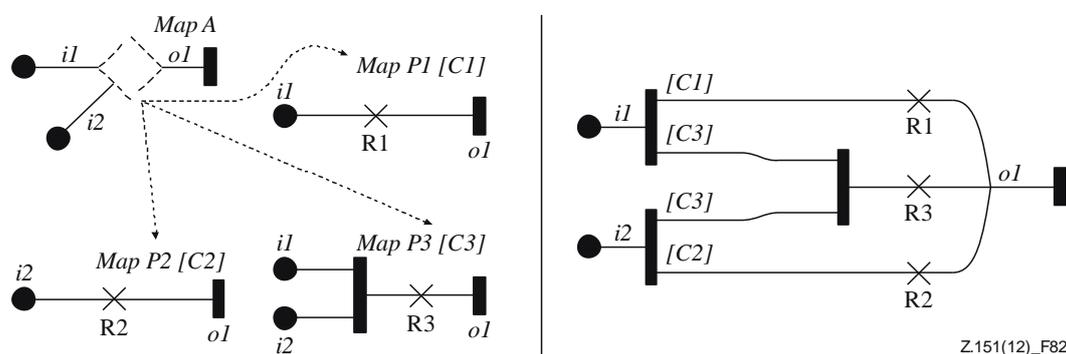


Figure 81 – Example: UCM plug-in map runtime instances

### iii) Flattened UCM models

The flattening of a static stub and its plug-in map is as follows. The in-paths of the stub are merged with the start points on the plug-in map according to the specified *InBindings*. The out-paths of the stub are merged with the end points on the plug-in map according to the specified *OutBindings*. The structural specifications on the parent map are merged with the structural specifications on the plug-in map according to the specified *ComponentBindings*. Structural specifications are treated the same way as for all types of stubs.

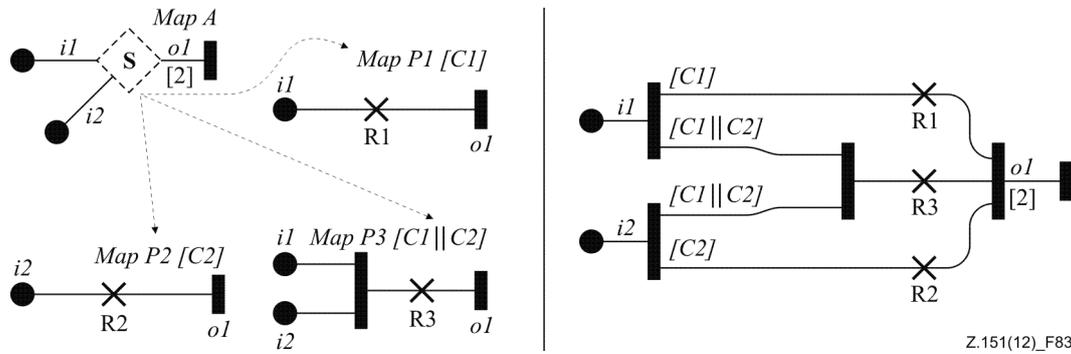
The semantics for a dynamic stub is similar to static stubs in that a dynamic stub shall contain only one runtime instance of each of its plug-in maps at a time. The semantics differs as the purpose of a dynamic stub is to model AND-forks and OR-joins in addition to simple hierarchical structuring (see Figure 82). Each in-path is equivalent to an AND-fork that is connected to the flattened plug-in maps according to the specified *InBindings*. Analogously, each out-path corresponds to an OR-join connected to the flattened plug-in maps based on the specified *OutBindings*. Plug-in bindings are indicated in the example below by labelling in-paths, out-paths, start points and end points with *iN* and *oN*. Preconditions of plug-in maps are indicated in square brackets next to the name of the plug-in map.



**Figure 82 – Example: Semantic flattening of a dynamic stub**

The semantics of an AND-fork in a flattened UCM model corresponds to the semantics of stubs with guarded plug-in maps and not the semantics of regular AND-forks in non-flattened models, i.e., guards on concurrent branches of an AND-fork are allowed. The URN metamodel accommodates these guards only for the purpose of flattened UCM models. AND-forks with guards, however, are not used in standard URN models.

The semantics for a synchronizing stub in terms of runtime instances of maps, however, is slightly different from the semantics for static and dynamic stubs, because the plug-in maps bound to a synchronizing stub have to act as one group. If an in-path of the synchronizing stub is visited for a second time, a second group of plug-in maps have to be created. Therefore, synchronizing stubs can contain more than one runtime instance of a plug-in map at the same time. This behaviour, however, is equivalent to one runtime instance with tokens flowing between AND-forks and AND-joins that can only synchronize if they were created by an AND-fork during the same visit. The synchronizing stub is therefore still conceptually equivalent to its flattened counterpart (see Figure 83) with each in-path converted to an AND-fork and each out-path converted to an AND-join. The connections of the AND-fork and AND-join to the flattened plug-in maps are again based on the specified *InBindings* and *OutBindings*.



**Figure 83 – Example: Semantic flattening of a synchronizing stub**

The semantics of AND-forks in flattened UCM models is the same as for dynamic stubs explained earlier. The semantics of AND-joins corresponds to synchronizing stubs and not the regular AND-joins in non-flattened models. Thus, they allow the specification of synchronization thresholds. Again, the URN metamodel already allows for these thresholds for the purpose of flattening UCM models even though they are not used in standard URN models.

d) *Model*

None.

e) *Examples*

Multiple plug-in map runtime instances for a synchronizing stub with multiple in-paths are created as follows. If an in-path is visited for the first time after a different in-path was visited the first time, then no new plug-in map runtime instances need to be created, because both traversals belong to the same visit. See the UCMs in Figure 84 and the first, second and third column of Table 9 for an example. Given a synchronizing stub with three in-paths *i1*, *i2* and *i3*, two plug-in maps bound to the stub as indicated in the example below, and a traversal order of the in-paths (*i1*: 1st, 4th, 5th; *i2*: 2nd, 6th; *i3*: 3rd, 7th, 8th, 9th), there will be four visits where runtime instances of both plug-in maps P1 and P2 are created. The first runtime instances of P1 and P2 are created at the first traversal of in-path *i1*. The second and third traversals do not cause new runtime instances to be created because these in-paths have not yet been used for the first runtime instances, and hence are part of the same visit. The fourth traversal creates the second set of runtime instances (second visit) because in-path *i1* is traversed for the second time. The fifth traversal creates the third set of runtime instances because in-path *i1* is again traversed. The sixth and seventh traversals use the runtime instances of the plug-in maps that belong to the second visit because the events go to the longest-waiting runtime instance of a plug-in map. The eighth traversal uses the third set of runtime instances. Finally, the ninth traversal causes the fourth set of runtime instances to be created because in-path *i3* is traversed for the fourth time.

The traversal of in-paths is important for the creation of plug-in map runtime instances. The traversal of start points on the plug-in map, however, is not important except in the case of an unconnected start point on a non-singleton plug-in map. Such a start point may also cause the creation of a plug-in map runtime instance. Consequently, if the needed runtime instance has already been created by the arrival at the unconnected start point, the traversal of an in-path then shall not trigger the creation of a plug-in map runtime instance.

NOTE 1 – Such a start point does not exist in the plug-in maps in Figure 84 as all of the start points are connected to the stub via plug-in bindings.

Furthermore, if a replication factor is defined for a plug-in map, then as many runtime instances of the plug-in map as specified by the replication factor are created.

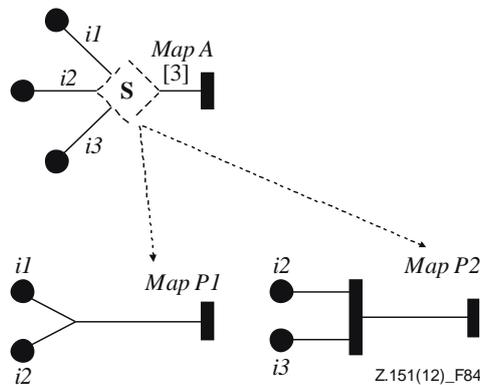
NOTE 2 – It is only possible to define a replication factor other than the default value for plug-in maps of dynamic stubs.

**Table 9 – Runtime instances and synchronizing stubs**

#	<i>In-path</i>	<i>Resulting action {specified synchronization threshold}</i>	<i>Resulting action {default synchronization threshold}</i>
1	i1	Create 1st P1 and 1st P2; continue with i1 on 1st P1	Create 1st P1 and 1st P2; set synchronization threshold to 2; continue with i1 on 1st P1
2	i2	Continue with i2 on 1st P1 (for the second time on that runtime instance) and 1st P2	Continue with i2 on 1st P1 (for the second time on that runtime instance) and 1st P2 {the synchronization threshold is reached and traversal continues past the stub}
3	i3	Continue with i3 on 1st P2 {the synchronization threshold is reached and traversal continues past the stub}	Continue with i3 on 1st P2 {ignore arrival at out-path}
4	i1	Create 2nd P1 and 2nd P2; continue with i1 on 2nd P1	Create 2nd P1 and 2nd P2; set synchronization threshold to 2; continue with i1 on 2nd P1
5	i1	Create 3rd P1 and 3rd P2; continue with i1 on 3rd P1	Create 3rd P1 and 3rd P2; set synchronization threshold to 2; continue with i1 on 3rd P1
6	i2	Continue with i2 on 2nd P1 (for the second time on that runtime instance) and 2nd P2	Continue with i2 on 2nd P1 (for the second time on that runtime instance) and 2nd P2 {the synchronization threshold is reached and traversal continues past the stub}
7	i3	Continue with i3 on 2nd P2 {the synchronization threshold is reached and traversal continues past the stub}	Continue with i3 on 2nd P2 {ignore arrival at out-path}
8	i3	Continue with i3 on 3rd P2	Continue with i3 on 3rd P2
9	i3	Create 4th P1 and 4th P2; continue with i3 on 4th P2	Create 4th P1 and 4th P2; set synchronization threshold to 2; continue with i3 on 4th P2

If the synchronization threshold is not specified in the example in Figure 84, then the default behaviour stipulates that as many plug-in map runtime instances shall arrive at the out-path as are traversed in parallel before the traversal is allowed to continue. The fourth column in Table 9 explains the behaviour of the synchronizing stub in this case assuming that both plug-in maps are selected.

The synchronization threshold is always specified upon first arrival at a stub during each visit. Subsequent arrivals during the same visit along other in-paths do not change the synchronization threshold for that visit, even if the number of plug-in map runtime instances that are being traversed changes.



**Figure 84 – Example: UCM synchronizing stub with threshold**

### 8.3.2 PluginBinding

*PluginBinding* defines the binding (i.e., connection) of behavioural and structural specifications on a parent map to behavioural and structural specifications on a plug-in map with the help of *ComponentBindings* (covered in clause 8.4), *InBindings* and *OutBindings*. A plug-in binding is contained by a stub and has a precondition that defines when the plug-in map is to be selected. Furthermore, a replication factor can be defined for a plug-in map, specifying how many runtime instances of the plug-in map are to be traversed in parallel. Finally, a plug-in map has a probability value stating the likelihood with which the plug-in map is selected in the UCM specification (see Figures 77 and 85).

#### a) Abstract grammar

##### i) Attributes

- id (String): The identifier of the plug-in binding.
- probability (Nat): The probability with which the plug-in map is selected in the UCM specification. Default value is 100.
- replicationFactor (String): The replication factor is an Integer expression that indicates how many runtime instances of the plug-in map are used.

##### ii) Relationships

- Contained by *Stub* (1): A *PluginBinding* is contained in one stub.
- Composition of *Condition* (0..1): A *PluginBinding* may contain one precondition.
- Composition of *InBinding* (0..\*): A *PluginBinding* may contain in-bindings.
- Composition of *OutBinding* (0..\*): A *PluginBinding* may contain out-bindings.
- Composition of *ComponentBinding* (0..\*): A *PluginBinding* may contain component bindings.
- Association with *UCMmap* (1): A *PluginBinding* has one plug-in map.

##### iii) Constraints

- id shall be unique within the URN specification.
- probability  $\geq 0$  and probability  $\leq 100$ .
- The replicationFactor shall be empty or an Integer expression, as defined in clause 9.3.
- The replicationFactor shall evaluate to a positive Integer value or it may be empty, in which case it is deemed to evaluate to 1.
- If the singleton attribute of the *UCMmap* of a *PluginBinding* its set to true, then the replicationFactor shall evaluate to 1.

- The *UCMmap* of a *PluginBinding* is the same as the *UCMmaps* that contain the *StartPoints* that belong to the *InBindings* of the *PluginBinding*.
- The *UCMmap* of a *PluginBinding* is the same as the *UCMmaps* that contain the *EndPoints* that belong to the *OutBindings* of the *PluginBinding*.
- The *UCMmap* containing the *Stub* of a *PluginBinding* is the same as the *UCMmaps* that contain the *NodeConnections* that belong to the *InBindings* of the *PluginBinding*.
- The *UCMmap* containing the *Stub* of a *PluginBinding* is the same as the *UCMmaps* that contain the *NodeConnections* that belong to the *OutBindings* of the *PluginBinding*.
- A *UCMmap* shall not be associated with more than one *PluginBinding* of the same *Stub*.

b) *Concrete grammar*

A *PluginBinding* does not have a graphical visual representation. Optionally, a *PluginBinding* is visualized next to its stub using the textual syntax for *PluginBinding* as specified in Annex B.

c) *Semantics*

*PluginBinding* groups together the *ComponentBindings*, *InBindings* and *OutBindings* of one plug-in map for one stub. The *Condition* of a plug-in map is a Boolean expression for a precondition that determines whether the plug-in map is selected when the traversal of a UCM path reaches the stub. If the plug-in map is selected, the traversal of the path continues on the plug-in map.

In addition, several attributes are defined by a *PluginBinding*. First, a *PluginBinding* is uniquely identified by its id.

Second, a *replicationFactor* defines for a plug-in map how many runtime instances of the plug-in map are to be traversed in parallel. A replication factor other than the default value may be defined for dynamic stubs but not for static stubs. Replicated maps for a dynamic stub are conceptually the same as copying one UCM map many times and plugging all copies with the same preconditions and the same bindings into the same stub.

Third, and for performance analysis purposes, a plug-in binding may have a probability which expresses the likelihood that the plug-in map is selected. The value of a probability is expressed relative to the probabilities of other plug-in maps of the same stub. A probability value in per cent is derived by dividing the value of the probability attribute by the sum of the probabilities of all plug-in maps of the same stub (i.e., 100 means that the plug-in map is selected, 0 means that the plug-in map is not selected, and 75 means that there is a 3:1 chance that the plug-in map is selected). Probabilities have no effect on the regular traversal of UCM models.

If no in-bindings are defined for a plug-in map or the precondition of the plug-in map evaluates to false, then the traversal of the UCM path stops at the stub on the parent map. If no out-bindings are defined for a plug-in map, the traversal of the UCM path stops at an end point on the plug-in map. If a condition is not specified for at least one alternative plug-in map (incompleteness), the traversal also stops and an error is generated.

### 8.3.3 InBinding

*InBinding* defines the connection of an in-path of a stub (i.e., a *NodeConnection*) on a parent map with a *StartPoint* on a plug-in map of the stub (see Figure 77).

a) *Abstract grammar*

i) **Attributes**

- None.

## ii) Relationships

- Contained by *PluginBinding* (1): An *InBinding* is contained in one plug-in binding.
- Association with *NodeConnection* (1): An *InBinding* consists of one node connection that represents an in-path of a stub.
- Association with *StartPoint* (1): An *InBinding* consists of one start point.

## iii) Constraints

- The target *PathNode* of the *NodeConnection* of an *InBinding* is the *Stub* that contains the *PluginBinding* of the *InBinding*.
- A *StartPoint* shall occur only once in all *InBindings* of a *PluginBinding*.
- Each in-path of a stub shall be in at least one *InBinding*.

### b) Concrete grammar

An *InBinding* does not have a graphical visual representation. Optionally, an *InBinding* is visualized as part of its *PluginBinding* using the textual syntax for *InBinding* as specified in Annex B.

### c) Semantics

The traversal of a UCM path utilizes the *InBindings* of a *Stub*'s plug-in map to move from the parent map to the plug-in map.

## 8.3.4 OutBinding

*OutBinding* defines the connection of an out-path of a stub (i.e., a *NodeConnection*) on a parent map with an *EndPoint* on a plug-in map of the stub (see Figure 77).

### a) Abstract grammar

#### i) Attributes

- None.

#### ii) Relationships

- Contained by *PluginBinding* (1): An *OutBinding* is contained in one plug-in binding.
- Association with *NodeConnection* (1): An *OutBinding* consists of one node connection that represents the out-path of a stub.
- Association with *EndPoint* (1): An *OutBinding* consists of one end point.

#### iii) Constraints

- The source *PathNode* of the *NodeConnection* of an *OutBinding* is the *Stub* that contains the *PluginBinding* of the *OutBinding*.
- An *EndPoint* shall occur only once in all *OutBindings* of a *PluginBinding*.
- Each out-path of a stub shall be in at least one *OutBinding*.

### b) Concrete grammar

An *OutBinding* does not have a graphical visual representation. Optionally, an *OutBinding* is visualized as part of its *PluginBinding* using the textual syntax for *OutBinding* as specified in Annex B.

### c) Semantics

The traversal of a UCM path utilizes the *OutBindings* of a *Stub*'s plug-in map to move from the plug-in map back to the parent map. The traversal shall only return to the same *Stub* through which the plug-in map was entered. The traversal, however, does not always return only one *Stub*. For example, if the traversal of two UCM paths entered a plug-in map through

two different *Stubs* S1 and S2 and the two UCM paths subsequently synchronize in the plug-in map and reach an end point E, then the traversal returns to S1 and S2 as long as an *OutBinding* exists from E to one out-path in each *Stub*.

## 8.4 UCM components

*Components* enable modelling of scenarios structure by specifying the entities involved in a scenario, covering the environment as well as the architectural structure of a system. Components may contain other components. Paths including any path node may be superimposed over components, thus allocating scenario behaviour to scenario structure. Map elements residing inside a component are said to be bound to the component. Components are characterized by a component kind and may also be typed. Component bindings belong to a plug-in binding and specify the relationship of components on a parent map with components on a plug-in map (see Figure 85).

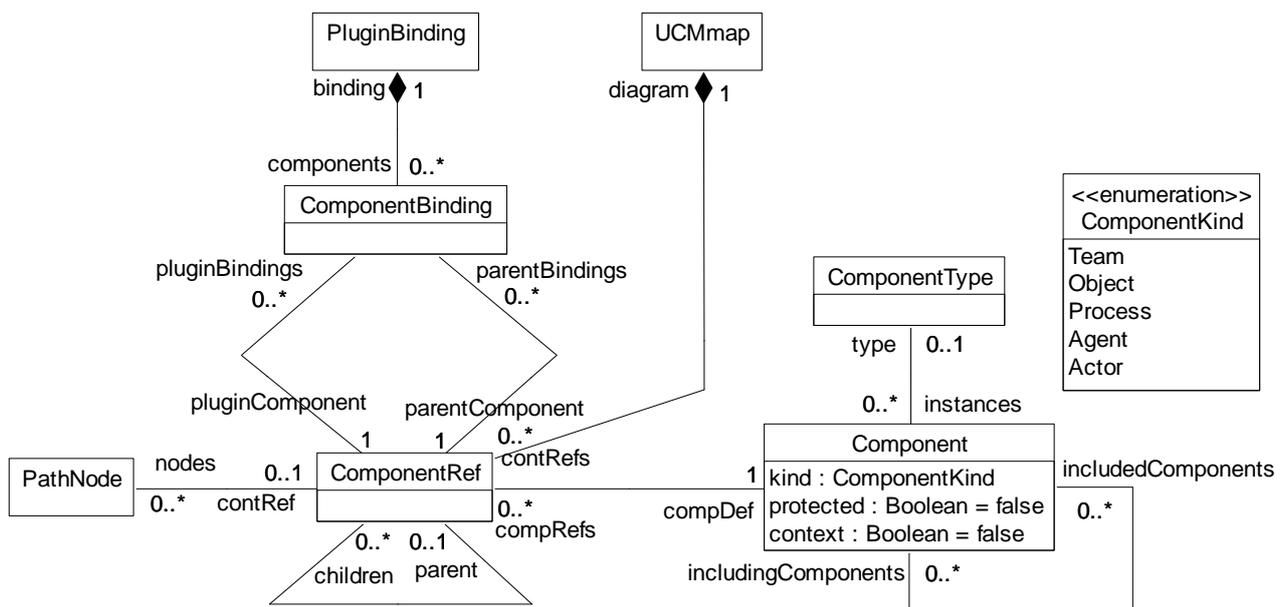


Figure 85 – Abstract grammar: UCM components

### 8.4.1 Component

A *Component* (also referred to as component definition) is a generic and abstract entity that can represent software entities (e.g., objects, processes, databases or servers) as well as non-software entities (e.g., actors or hardware). A component definition is characterized by its kind and its optional type, may contain other component definitions or be contained in other component definitions, may require the context of a parent scenario to be fully defined and may allow at the most one UCM path to be active inside its boundary at any time (i.e., a mutual exclusion mechanism) (see Figure 85).

a) *Abstract grammar*

#### i) Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 59).
- *kind* (*ComponentKind*): The kind of component. Default value is Team.
- *protected* (Boolean): Indicates whether the traversal of UCM paths allocated to the component definition is ruled by a mutual exclusion mechanism (true) or not (false). Default value is false (i.e., the component is not protected).
- *context* (Boolean): Indicates whether the component definition requires a component from a parent map to be connected to the component definition with the help of a component plug-in binding (true) or not (false). Default value is false.

## ii) Relationships

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *Component* definition is contained in the UCM specification (see Figure 58).
- Association with *ComponentType* (0..1): A *Component* definition may have one component type.
- Association with *Component* (includingComponents, 0..\*): A *Component* definition may be included by component definitions.
- Association with *Component* (includedComponents, 0..\*): A *Component* definition may include component definitions.
- Association with *ComponentRef* (0..\*): A *Component* definition may be referenced by component references.
- Association with *ProcessingResource* (0..1): A *Component* definition may be hosted by one processing resource (see Figure 94).
- Association with *PassiveResource* (0..1): A *Component* definition may correspond to one passive resource (see Figure 94).
- Uses *ComponentKind* enumeration.

## iii) Constraints

- Inherits constraints from *UCMmodelElement*.
- Any two *Component* definitions cannot share the same name inside a URN specification.
- The name of a *Component* definition cannot be the empty String "".
- The *Component* containment hierarchy established by the *includedComponents* relationship does not contain any cycles (i.e., a *Component* definition shall not appear more than once on a path from a top node to a leaf node in the containment hierarchy).
- The context attribute of the *Component* definition of a *pluginComponent* in a *ComponentBinding* shall be true.
- The kind attribute of the *Component* definition of a *pluginComponent* in a *ComponentBinding* shall be Team.
- The *Component* definition of a *pluginComponent* in a *ComponentBinding* shall not have a *ComponentType*.

## b) Concrete grammar

*Component* definition has no concrete syntax, but component references (see *ComponentRef*, clause 8.4.4) for the component definition are visualized. The line and fill colours of a component definition are specified in its definition's *ConcreteStyle* and are hence shared by all the component definition's references.

## i) Relationships

- Inherits relationships from *UCMmodelElement*.
- Composition of *ConcreteStyle* (0..1): A *Component* definition may have one concrete style (see Figure 55).

## c) Semantics

*Component* definitions represent the underlying structure of scenarios and may contain other components. A component definition may itself be contained in other components. The containment hierarchy of component definitions does not necessarily have to match the containment hierarchy of component references (see *ComponentRef* in clause 8.4.4) as intermediate *ComponentRefs* may not be shown on a UCM diagram. A component definition

may have a user-defined *ComponentType*, which further characterizes the component definition but does not influence the traversal of a UCM path bound to the component.

The traversal of a UCM path, however, is influenced by the kind of component as detailed in clause 8.4.3. The traversal also takes into account the *protected* attribute of a component definition. Upon entering a component reference associated with a protected component definition along a path, the traversal continues only if no other path is being traversed in the component definition.

The context attribute specifies that a *ComponentBinding* should exist for the component definition (i.e., when the component reference to the component definition is used on a plug-in map, then a binding to a component on the parent map should be specified). The existence of such a component binding, however, is not mandatory. If the traversal reaches a component reference to a component definition with the context attribute set to *true*, but no component binding is specified, then the traversal issues only a warning but continues the scenario.

A component definition may have several includingComponents (i.e., more than one parent), therefore allowing the capture of several architectural alternatives in one UCM model. A modeller may investigate various allocations of sub-components to components, usually defined in different plug-in maps of a dynamic stub, and reason about trade-offs involving these alternatives. The alternatives may also be reasoned about and evaluated more formally in the URN model with the help of GRL models for the alternative component structures.

Component definitions also play a role in the performance analysis of UCM specifications as explained in clause 8.6. A component definition can be optionally hosted on a *ProcessingResource*, which then becomes the target of host demands made by responsibility references bound to references to that component definition. A component definition may optionally be considered as a *PassiveResource*.

**NOTE – Semantic variation:** Modellers are allowed to impose additional constraints on the containment hierarchy of component definitions. For example, if the specification of architectural alternatives in one UCM model is not desired, a component definition is allowed to be contained at most in only one parent component definition. Furthermore, additional constraints could be imposed in terms of how different kinds of components may be contained in other components. For example, a component of kind *Process* is not allowed to be contained in a component of kind *Object*. Such constraints could also be extended to user-defined *ComponentTypes*, so that:

- A *Component* is only allowed to be included at the most by one other *Component*, and
- Let C1 and C2 be *Component* definitions and the kind of C1 be set to *Object*. If C1 is the ancestor of C2 in the containment hierarchy of *Component* definitions established by the *includedComponents* relationship of *Component* definitions, then the kind of C2 is not allowed to be set to *Process*.

## 8.4.2 ComponentType

A *ComponentType* allows the definition of user-defined types of components (see Figure 85).

a) *Abstract grammar*

### i) Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 59).

### ii) Relationships

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *ComponentType* is contained in the UCM specification (see Figure 58).
- Association with *Component* (0..\*): A *ComponentType* may be assigned to component definitions.

### iii) Constraints

- Inherits constraints from *UCMmodelElement*.

### b) Concrete grammar

A *ComponentType* does not have a graphical visual representation. However, it influences the presentation of components references (see clause 8.4.4).

### i) Relationships

- Inherits relationships from *UCMmodelElement*.

### c) Semantics

*ComponentTypes* group *Component* definitions according to a user-defined name. For example, a call scenario may include two call agents, one for subscriber A and one for subscriber B. While the names of the component definitions for these call agents are "subscriber A" and "subscriber B", the name of the component type is "call agent" and is associated with both component definitions. The component type, however, does not influence the traversal of UCM paths.

## 8.4.3 ComponentKind

A component definition can be a Team, a Process, an Object, an Agent or an Actor (see Figure 85).

### a) Abstract grammar

#### i) Attributes

- None (enumeration metaclass).

#### ii) Relationships

- Used by *Component*.

#### iii) Constraints

- None.

### b) Concrete grammar

None (enumeration metaclass). However, it influences the presentation of components references (see clause 8.4.4).

### c) Semantics

*ComponentKind* differentiates between several kinds of components. A Team is a generic component, used as a container for sub-components of any kind. A Process is an active component, which implies the existence of a control thread. An Object is a passive component, which is usually controlled by a process. An Agent is an autonomous component, which acts on behalf of other components. An Actor is an external component that describes an entity, either human or artificial, that interacts with the system.

The traversal of UCM paths treats all kinds of components the same way with the exception of components of kind Object. The traversal interleaves the traversal of path nodes of parallel branches that are bound to the same component definition, if the component definition is of kind Object.

## 8.4.4 ComponentRef

*ComponentRef* references a component definition. In a URN specification, the same component definition may be referenced many times in the same UCM diagram and in many UCM diagrams. Component references may contain other component references and path nodes. Relationships between component references on a parent map and component references on a plug-in map may be established with the help of component plug-in bindings (see Figure 85).

a) *Abstract grammar*

i) **Attributes**

- Inherits attributes from *UCMmodelElement* (see Figure 59).

ii) **Relationships**

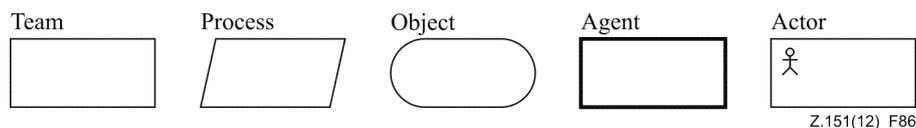
- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMmap* (1): A *ComponentRef* is contained in one UCM map.
- Association with *PathNode* (0..\*): A *ComponentRef* may contain path nodes.
- Association with *Component* (1): A *ComponentRef* references one component definition.
- Association with *ComponentBinding* (parentBindings, 0..\*): A *ComponentRef* may be the parent component in component bindings.
- Association with *ComponentBinding* (pluginBindings, 0..\*): A *ComponentRef* may be the component on the plug-in map in component bindings.
- Association with *ComponentRef* (parent, 0..1): A *ComponentRef* may be included by one component reference.
- Association with *ComponentRef* (children, 0..\*): A *ComponentRef* may include component references.

iii) **Constraints**

- Inherits constraints from *UCMmodelElement*.
- The name of a *ComponentRef* shall be the same as the name of its associated *Component* definition.
- The *UCMmap* that contains the *ComponentRef* shall be the *UCMmap* that contains *ComponentRefs* associated as children.
- The containment hierarchy of *ComponentRefs* established by the children relationship does not contain any cycles (i.e., a *ComponentRef* shall not appear more than once on a path from a top node to a leaf node in the containment hierarchy).
- Let CR1 and CR2 be *ComponentRefs*. If CR1 is the parent of CR2, then the *Component* definition of CR1 shall be an ancestor of the *Component* definition of CR2 in the containment hierarchy of *Component* definitions established by the includedComponents relationship of *Component* definitions.

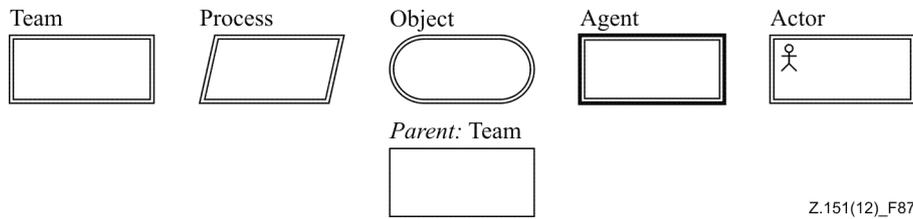
b) *Concrete grammar*

The symbol for *ComponentRef* on a UCM map depends on the kind of the *Component* definition to which the *ComponentRef* refers: a rectangle for Team, a parallelogram for Process, a rounded-corner rectangle for Object, a rectangle with a thick border for Agent and a rectangle with a stickman icon in its top-left corner for Actor (see Figure 86, where the name of each component reference is the same as its kind, for illustration purposes).



**Figure 86 – Symbol: UCM component reference**

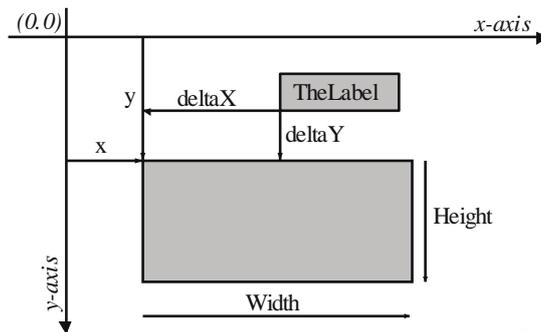
If the protected attribute of the *Component* definition to which the *ComponentRef* refers is true, then a second, slightly smaller outline is added to the symbol for *ComponentRef* as shown in Figure 87.



Z.151(12)\_F87

**Figure 87 – Symbol: UCM protected and context-dependent component reference**

The name of the component reference (from superclass *URNmodelElement*) is displayed next to the symbol according to *Label* coordinates. Optionally, the *ComponentType* of the *Component* to which the *ComponentRef* refers is shown in parentheses next to the name of the component reference. If the context attribute of the *Component* definition to which the *ComponentRef* refers is true, then the name is prefixed with "parent:" in italic font (see Figure 87). The coordinate conventions of clause 5.3.2 apply. The top-left corner of the *ComponentRef* is indicated by its *Position* (x, y) and the bottom-right corner by its *Position* and *Size* (x+width, y+height). The bottom-left corner of the *Label* is relative to the *Position* (x-deltaX, y-deltaY) (see Figure 88) for an illustration of these layout principles.



Z.151(12)\_F88

**Figure 88 – Layout: Position, size and label for ComponentRef**

**i) Relationships**

- Inherits relationships from *UCMmodelElement*.
- Composition of *Label* (0..1): A *ComponentRef* may have one label (see Figure 52).
- Composition of *Size* (0..1): A *ComponentRef* may have one size (see Figure 54).
- Composition of *Position* (0..1): A *ComponentRef* may have one position (see Figure 53).

**ii) Constraints**

- A *ComponentRef* shall have one *Label*.
- A *ComponentRef* shall have one *Size*.
- A *ComponentRef* shall have one *Position*.
- The symbol of a *ComponentRef* shall not overlap with the symbol of another *ComponentRef*, unless it is entirely inside the symbol of that *ComponentRef*.

c) *Semantics*

*ComponentRef* allows for the reuse of the same *Component* definition in multiple locations on one or more *UCMmaps*. The semantics of *ComponentRef* is defined by the attributes of its *Component* definition and its *ComponentBindings* (see clauses 8.4.1 and 8.4.5, respectively).

d) *Model*

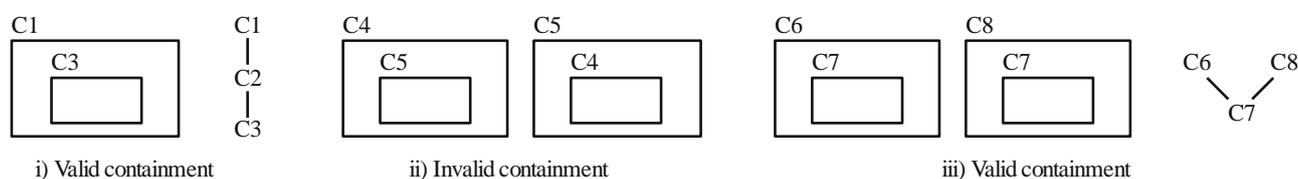
None.

e) *Examples*

The following examples demonstrate the relationship of component definitions and component references in terms of component containment. First of all, if *Component* definition C1 is the parent of *Component* definition C2, which in turn is the parent of *Component* definition C3, then a UCM diagram may show a *ComponentRef* to C3 being contained in a *ComponentRef* to C1 without showing a *ComponentRef* for the intermediary C2. This is possible because constraints 8.4.1d and 8.4.4e stipulating that containment hierarchies shall be cycle-free and that C1 shall be an ancestor of C3 in the containment hierarchy is not violated (see Figure 89.i).

If, however, a *ComponentRef* to C4 is the parent of a *ComponentRef* to C5 and another *ComponentRef* to C5 is the parent of another *ComponentRef* to C4, then the constraints are violated, because an attempt to satisfy both *ComponentRef* containments leads to a cycle in the containment hierarchy (see Figure 89.ii).

On the other hand, if a *ComponentRef* to C6 is the parent of a *ComponentRef* to C7 and a *ComponentRef* to C8 is the parent of another *ComponentRef* to C7, then the constraints are not violated. In this case, however, the containment hierarchy for *Component* definitions is not yet fully specified as there are several possible options for the parent of C7. First, C6 is the parent of C8, which in turn is the parent of C7. Second, C8 is the parent of C6, which in turn is the parent of C7. Third, the UCM model may specify structural alternatives and therefore only one of C6 and C8 is the parent of C7 but there is no containment relationship between C6 and C8. The third option is the most general interpretation of the hierarchy of component references and is therefore reflected in the containment hierarchy of component definitions (see Figure 89.iii).



Z.151(12)\_F89

**Figure 89 – Examples: UCM component containment hierarchies**

### 8.4.5 ComponentBinding

*ComponentBinding* captures the relationship of component references on a parent map with components references on a plug-in map (see Figure 85).

a) *Abstract grammar*

i) **Attributes**

– None.

## ii) Relationships

- Contained by *PluginBinding* (1): A *ComponentBinding* is contained in one plug-in binding.
- Association with *ComponentRef* (parentComponent, 1): A *ComponentBinding* consists of one component reference on the parent map.
- Association with *ComponentRef* (pluginComponent, 1): A *ComponentBinding* consists of one component reference on the plug-in map.

## iii) Constraints

- The *UCMmap* of the *ComponentRef* associated as parentComponent is the *UCMmap* of the *Stub* of the *PluginBinding* of the *ComponentBinding*.
- The *UCMmap* of the *ComponentRef* associated as pluginComponent is the *UCMmap* of the *PluginBinding* of the *ComponentBinding*.
- Let M be a non-singleton *UCMmap* and let PG be a pluginComponent group where each pluginComponent is contained in M, has the same *Component* definition, and belongs to a *ComponentBinding* in the same *PluginBinding*. Then, each *Component* definition of a parentComponent of a *ComponentBinding* of the pluginComponents in PG shall be the same.
- Let M be a singleton *UCMmap* and let PG be a pluginComponent group where each pluginComponent is contained in M, has the same *Component* definition, and belongs to a *ComponentBinding* in any *PluginBinding*. Then, each *Component* definition of a parentComponent of a *ComponentBindings* of the pluginComponents in PG shall be the same.

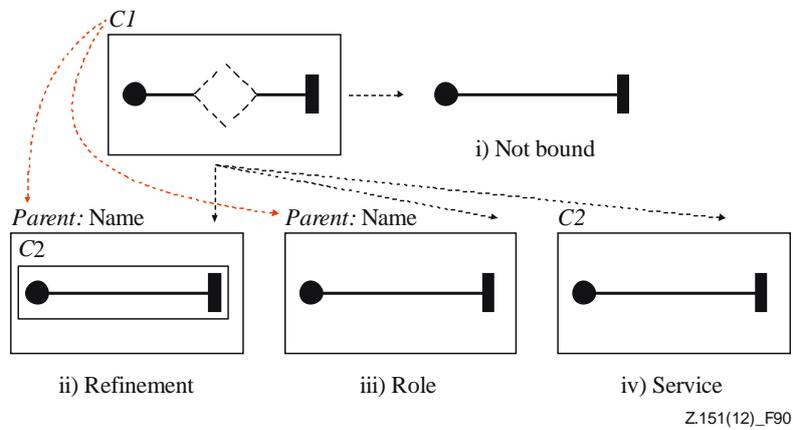
### b) Concrete grammar

A *ComponentBinding* does not have a graphical visual representation. Optionally, a *ComponentBinding* is visualized as part of its *PluginBinding* using the textual syntax for *ComponentBinding* as specified in Annex B.

### c) Semantics

*ComponentBinding* establishes a relationship between a component reference on a parent map with a component reference on a plug-in map. When a component reference with a component plug-in binding is reached on a plug-in map during the traversal of a UCM path, the component definition of the component reference on the plug-in map is not used by the traversal. Instead, the traversal of a UCM path uses the component definition of the parent component reference. Consequently, the component definition of the component reference on the plug-in map is irrelevant. Therefore, its component kind and type are also irrelevant and not specified.

Component bindings may specify four different relationships as illustrated in the examples in Figure 90. The path on the plug-in map may not be bound to a component reference at all (i), the structure of the parent component C1 is refined on the plug-in map as component C2 is contained in the parent component (ii), the parent component C1 is playing a role specified on the plug-in map (e.g., a role in an architectural or behavioural pattern) (iii), and the parent component C1 uses the services provided by component C2 as specified by the path bound to C2 on the plug-in map (iv). Consequently, the location of a stub relative to components on the parent map does not have any semantic significance (i.e., the behaviour and structure defined on a plug-in map of a stub do not necessarily have to be bound to the same component reference as the stub).



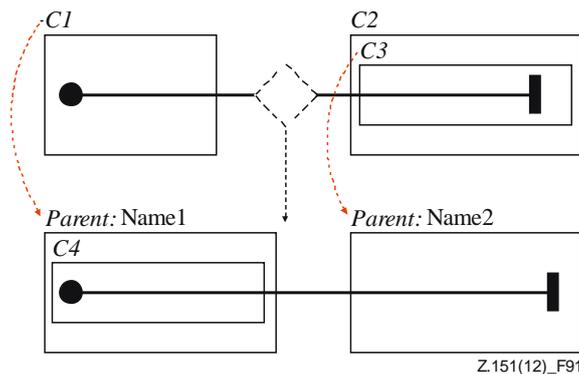
**Figure 90 – Example: Plug-in bindings for components**

d) *Model*

None.

e) *Example*

With the help of plug-in bindings for components, the relationship of multiple components on a parent map and on plug-in maps may also be modelled as shown in the example in Figure 91. Components on the plug-in map for which plug-in bindings are supposed to exist are identified as usual by the prefix "parent:". Parent component C1 is bound to component Name1 on the plug-in map, whereas parent component C3 is bound to component Name2 on the plug-in map. This example also shows that component plug-in bindings may be established regardless of the location of the stub on the parent map, i.e., even if the stub is not bound to any component reference on the parent map.

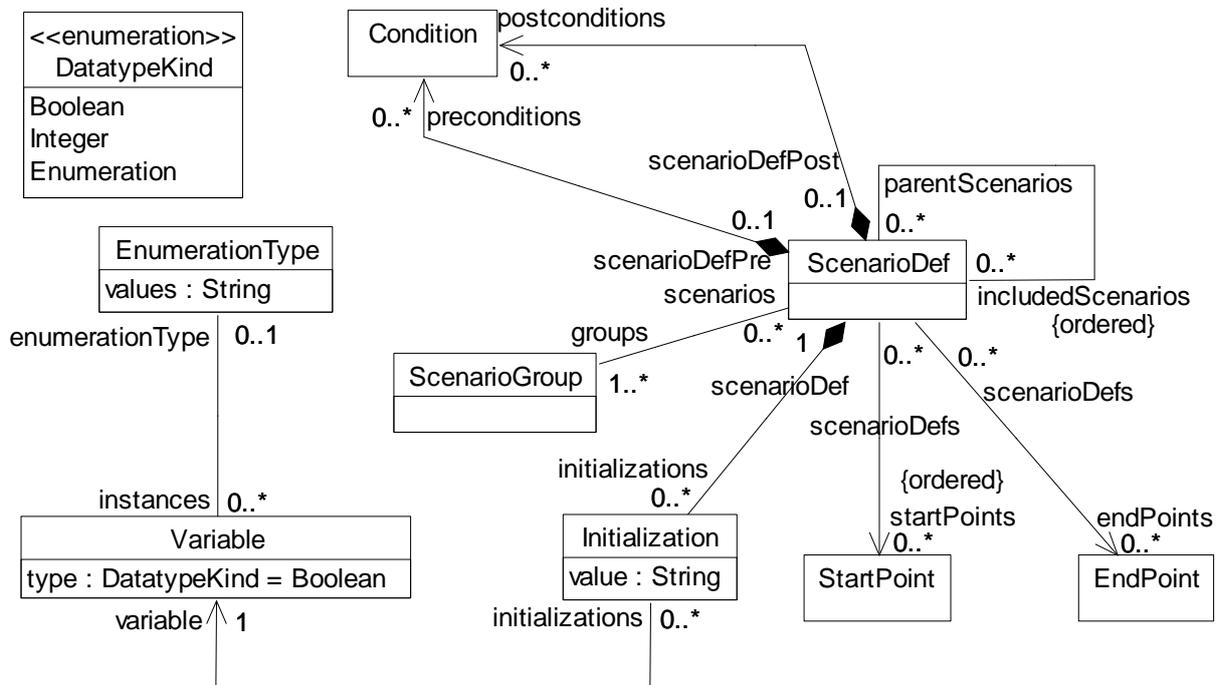


**Figure 91 – Example: Plug-in bindings for multiple components**

## 8.5 UCM scenario definitions

Scenario definitions (see *ScenarioDef*, clause 8.5.2) make use of path variables and conditions to identify individual scenarios in an integrated collection of UCMs. Conditions allow the explicit definition of otherwise hidden causal dependencies of path segments, thereby reducing the number of path segments that can be combined to create useful and sensible end-to-end scenarios. Once defined, such scenarios can be grouped or used for highlighting and animating specific paths or for generating other representations such as Message Sequence Charts or TTCN-3 test cases (see Figure 92).

There are no specific concrete grammar metaclasses for the model elements defined in this clause.



**Figure 92 – Abstract grammar: UCM scenario definitions**

### 8.5.1 ScenarioGroup

*ScenarioGroup* is a collection of scenario definitions. It is used to organize scenario definitions and to manipulate them as a group (see Figure 92).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *UCMmodelElement* (see Figure 59).

**ii) Relationships**

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *ScenarioGroup* is contained in the UCM specification (see Figure 58).
- Association with *ScenarioDef* (0..\*): A *ScenarioGroup* may refer to scenario definitions.

**iii) Constraints**

- Inherits constraints from *UCMmodelElement*.

b) *Concrete grammar*

A *ScenarioGroup* does not have a graphical visual representation. Optionally, a *ScenarioGroup* is visualized using the textual syntax for *ScenarioGroup* as specified in Annex B.

**i) Relationships**

- Inherits relationships from *UCMmodelElement*.

c) *Semantics*

None (*ScenarioGroup* is a structural concept only).

## 8.5.2 ScenarioDef

*ScenarioDef* defines a scenario through the UCM model (i.e., a path through the model for which alternatives at each choice point have been chosen). A scenario definition includes the start points of the scenario, the desired end points to be reached, preconditions and postconditions that have to be satisfied and initialization values for variables in the global data model of the URN specification (see clause 9.1) (see Figure 92).

### a) *Abstract grammar*

#### i) **Attributes**

- Inherits attributes from *UCMmodelElement* (see Figure 59).

#### ii) **Relationships**

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *ScenarioDef* is contained in the UCM specification (see Figure 58).
- Composition of *Condition* (preconditions, 0..\*): A *ScenarioDef* may contain preconditions.
- Composition of *Condition* (postconditions, 0..\*): A *ScenarioDef* may contain postconditions.
- Composition of *Initialization* (0..\*): A *ScenarioDef* may contain variable initializations.
- Association with *ScenarioGroup* (1..\*): A *ScenarioDef* is referenced by at least one group of scenarios.
- Association with *ScenarioDef* (parentScenarios, 0..\*): A *ScenarioDef* may be included by scenario definitions.
- Association with *ScenarioDef* (includedScenarios, 0..\*) {ordered}: A *ScenarioDef* may have an ordered collection of included scenario definitions.
- Association with *StartPoint* (0..\*) {ordered}: A *ScenarioDef* may define an ordered collection of start points to be triggered.
- Association with *EndPoint* (0..\*): A *ScenarioDef* may define end points to be reached.

#### iii) **Constraints**

- Inherits constraints from *UCMmodelElement*.
- The scenario containment hierarchy established by the *includedScenarios* relationship does not contain any cycles (i.e., a *ScenarioDef* shall not appear more than once on a path from a top node to a leaf node in the containment hierarchy).

### b) *Concrete grammar*

A *ScenarioDef* does not have a graphical visual representation. Optionally, a *ScenarioDef* is visualized using the textual syntax for *ScenarioDef* as specified in Annex B.

#### i) **Relationships**

- Inherits relationships from *UCMmodelElement*.

### c) *Semantics*

*ScenarioDef* is used as the starting point for the UCM traversal mechanism. The traversal of a path begins at the first *StartPoint* specified in a scenario definition, if the preconditions (see *Condition*, clause 6.1.6) are satisfied after initializing the variables in the global data model of the URN specification according to the *Initializations* specified in the scenario definition. If a precondition is not satisfied, the traversal of the scenario failed and an error is generated.

The start points are triggered in the order specified in the scenario definition, beginning with the first one. The following start point is only triggered when the traversal triggered by the first start point cannot proceed any further. If an error occurs, none of the remaining start points are triggered.

At the end of the traversal of a path, the reached *EndPoints* are compared against the desired *EndPoints* specified in the scenario definition and the postconditions of the scenario definition are evaluated. Postconditions of scenario definitions may make use of variable names with the suffix "\_pre", denoting the value of a variable *Initialization*. The scenario traversal completed successfully if all desired end points have been reached and all postconditions evaluate to true. In all other cases, the traversal of the scenario failed and an error is generated.

If a scenario definition contains another scenario definition, the traversal of a UCM path considers the union of the *StartPoints*, *EndPoints*, preconditions and postconditions for its purposes. For conflicting *Initializations* (i.e., initializations of the same variable), however, the *Initialization* of the containing scenario overrides the *Initialization* of the contained scenario. If more than one scenario definition is included, then the last scenario definition in the ordered list of included scenarios takes precedence over the previous ones (i.e., the initializations are applied beginning with the first included scenario definition in the ordered list; then each other scenario definition is applied up to the last included scenario definition, and finally the initializations of the including scenario are applied).

If the preconditions of a scenario definition contradict the preconditions defined by its contained scenario definitions, then the traversal will never be able to start. Analogously for postconditions, the traversal will never be able to finish successfully if postconditions contradict each other. The order of the start points in the union of start points from all included scenarios and the scenario itself is determined by the ordered list of included scenarios defined in a scenario definition. The start points of the first included scenario definition have priority over all other start points, followed by the next included scenario definition, and eventually the last included scenario definition. The start points of the scenario definition itself are triggered last. Within each set of start points, the start points are again ordered and the first start point has the highest priority.

d) *Model*

None.

e) *Examples*

Scenario definitions can help identify problems with a UCM specification as they can be used to specify desired scenario behaviour. Essentially, scenario definitions are test cases that can be run by the UCM path traversal against a UCM specification. The example in this clause builds on the example UCM model from clause 8.2.1, illustrates different ways of structuring scenario definitions, and shows how the UCM path traversal may help detect undesired interactions between scenarios.

One approach to structuring scenario definitions defines complete end-to-end scenarios for each scenario definition. End-to-end scenarios may consist of many features defined on separate plug-in maps as shown in Figure 61. With this approach, a scenario definition does not include other scenario definitions but specifies all required scenario definition elements itself. For example, the scenario definitions in Table 10 are structured according to this approach.

**Table 10 – Example: End-to-end UCM scenario definitions**

<i>Name</i>	<i>Start points</i>	<i>Initializations<sup>1)</sup></i>	<i>End points</i>
Basic Call success	request	!subOCS, !subTL, !subTCS, !busy	ring, ringing
Basic Call busy	request	!subOCS, !subTL, !subTCS, busy	busy
OCS success	request	subOCS, !subTL, !subTCS, !busy, !onOCslist	ring, ringing
OCS busy	request	subOCS, !subTL, !subTCS, busy, !onOCslist	busy
OCS denied	request	subOCS, !subTL, !subTCS, onOCslist	notify
TL success	request	!subOCS, subTL, !subTCS, !busy, !TLactive	ring, ringing
TL pin success	request, enterPIN	!subOCS, subTL, !subTCS, !busy, TLactive, PINvalid	ring, ringing
TL busy	request	!subOCS, subTL, !subTCS, busy, !TLactive	busy
TL pin busy	request, enterPIN	!subOCS, subTL, !subTCS, busy, TLactive, PINvalid	busy
TL invalid pin	request, enterPIN	!subOCS, subTL, !subTCS, !busy, TLactive, !PINvalid	notify
TL timeout	request	!subOCS, subTL, !subTCS, !busy, TLactive	notify
TCS success	request	!subOCS, !subTL, subTCS, !busy, !onTCSlist	ring, ringing
TCS busy	request	!subOCS, !subTL, subTCS, busy, !onTCSlist	busy
TCS denied	request	!subOCS, !subTL, subTCS, onTCSlist	notify
TL pin TCS success	request, enterPIN	!subOCS, subTL, subTCS, !busy, TLactive, PINvalid, !onTCSlist	ring, ringing

<sup>1)</sup> <variable> is shorthand for <variable> = true; !<variable> is shorthand for <variable> = false.

The first two scenario definitions specify basic call behaviour, the next three the OCS feature combined with basic call, the next six the TL feature combined with basic call, and the next three feature the TCS feature combined with basic call. The last scenario definition specifies the behaviour of a combination of TL, TCS and basic call.

Another way of structuring scenario definitions is to use one scenario definition per feature that captures the common specifications for all basic scenario definitions of the feature. For example, the scenario definitions in Table 11 show a common scenario definition used by the TL feature.

**Table 11 – Example: End-to-end UCM scenario definitions with common elements**

<i>Name</i>	<i>Start points</i>	<i>Initializations<sup>1)</sup></i>	<i>End points</i>
TL common	request	!subOCS, subTL, !subTCS, !busy, TLactive, PINvalid	
The following scenario definitions include TL common (elements that are scenario definition-specific and not defined in TL common are shown in <b><i>bold</i></b> and <i>italic</i> ):			
TL success	request	!subOCS, subTL, !subTCS, !busy, <b><i>!TLactive</i></b> , PINvalid	<b><i>ring, ringing</i></b>
TL pin success	request, <b><i>enterPIN</i></b>	!subOCS, subTL, !subTCS, !busy, TLactive, PINvalid	<b><i>ring, ringing</i></b>
TL busy	request	!subOCS, subTL, !subTCS, <b><i>busy</i></b> , <b><i>!TLactive</i></b> , PINvalid	<b><i>busy</i></b>

**Table 11 – Example: End-to-end UCM scenario definitions with common elements**

<i>Name</i>	<i>Start points</i>	<i>Initializations<sup>1)</sup></i>	<i>End points</i>
TL pin busy	request, <i>enterPIN</i>	!subOCS, subTL, !subTCS, <i>busy</i> , TLactive, PINvalid	<i>busy</i>
TL invalid pin	request, <i>enterPIN</i>	!subOCS, subTL, !subTCS, ! <i>busy</i> , TLactive, <i>!PINvalid</i>	<i>notify</i>
TL timeout	request	!subOCS, subTL, !subTCS, ! <i>busy</i> , TLactive, PINvalid	<i>notify</i>
<sup>1)</sup> <variable> is shorthand for <variable> = true; !<variable> is shorthand for <variable> = false.			

The elements from the common scenario definition are merged with the elements from the including scenario definition. The including scenario definition may override initializations of the common scenario definition (e.g., TL success and TL busy override TLactive, TL busy and TL pin busy override busy, and TL invalid pin overrides PINvalid). This second approach of structuring scenario definitions demonstrates one reason for providing included scenarios in the abstract UCM metamodel. Common prefixes of a scenario can be captured and reused across many scenario definitions. By including a scenario definition, a scenario is positioned at the desired path location. For example, any TL scenario always proceeds from the request start point to the OrigFeatures stub. Any TCS scenario, on the other hand, always proceeds from the request start point to the TermFeatures stub.

A third way of structuring scenario definitions defines feature-specific scenario definitions that do not describe end-to-end scenarios. An end-to-end scenario is created by including several scenario definitions in another scenario definition. Table 12 gives an example of this approach with the help of the basic call, OCS, TL and TCS features. Elements of a scenario definition that are scenario definition-specific and not defined in an included scenario definition are shown in *bold* and *italic*. The basic feature-specific scenarios are highlighted in **bold**.

**Table 12 – Example: Feature-specific UCM scenario definitions**

<i>Name<sup>1)</sup></i>	<i>Start points</i>	<i>Initializations<sup>2)</sup></i>	<i>End points</i>
<b>Basic Call core</b>	<i>request</i>	<i>!subOCS, !subTL, !subTCS, !busy</i>	
Basic Call success +Basic Call core	request	!subOCS, !subTL, !subTCS, !busy	<i>ring, ringing</i>
Basic Call busy +Basic Call core	request	!subOCS, !subTL, !subTCS, <i>busy</i>	<i>busy</i>
<b>OCS core</b>		<i>subOCS, !onOCSlist</i>	
OCS success +Basic Call success, +OCS core	request	subOCS, !subTL, !subTCS, !busy, !onOCSlist	ring, ringing
OCS busy +Basic Call busy, +OCS core	request	subOCS, !subTL, !subTCS, busy, !onOCSlist	busy
OCS denied +Basic Call core +OCS core	request	subOCS, !subTL, !subTCS, <i>onOCSlist</i>	<i>notify</i>
<b>TL core</b>		<i>subTL, TLactive, PINvalid</i>	

**Table 12 – Example: Feature-specific UCM scenario definitions**

<i>Name<sup>1)</sup></i>	<i>Start points</i>	<i>Initializations<sup>2)</sup></i>	<i>End points</i>
<b>TL core PIN</b> +TL core	<i>enterPIN</i>	subTL, TLactive, PINvalid	
TL success +Basic Call success +TL core	request	!subOCS, subTL, !subTCS, !busy, <b>!TLactive</b> , PINvalid	ring, ringing
TL pin success +Basic Call success +TL core PIN	request, enterPIN	!subOCS, subTL, !subTCS, !busy, TLactive, PINvalid	ring, ringing
TL busy +Basic Call busy +TL core	request	!subOCS, subTL, !subTCS, busy, <b>!TLactive</b> , PINvalid	busy
TL pin busy +Basic Call busy +TL core PIN	request, enterPIN	!subOCS, subTL, !subTCS, busy, TLactive, PINvalid	busy
TL invalid pin +Basic Call core +TL core PIN	request, enterPIN	!subOCS, subTL, !subTCS, !busy, TLactive, <b>!PINvalid</b>	<b>notify</b>
TL timeout +Basic Call core +TL core	request	!subOCS, subTL, !subTCS, !busy, TLactive, PINvalid	<b>notify</b>
<b>TCS core</b>		<b>subTCS, !onTCSlist</b>	
TCS success +Basic Call success +TCS core	request	!subOCS, !subTL, subTCS, !busy, !onTCSlist	ring, ringing
TCS busy +Basic Call busy +TCS core	request	!subOCS, !subTL, subTCS, busy, !onTCSlist	busy
TCS denied +Basic Call core +TCS core	request	!subOCS, !subTL, subTCS, <b>onTCSlist</b>	<b>notify</b>
Below are scenario definitions that combine more than two features:			
TL pin TCS success +Basic Call success +TL core PIN +TCS core	request, enterPIN	!subOCS, subTL, subTCS, !busy, TLactive, PINvalid, !onTCSlist	ring, ringing
TL invalid pin OCS +Basic Call core +TL core PIN +OCS core	request, enterPIN	subOCS, subTL, !subTCS, !busy, TLactive, <b>!PINvalid</b> , !onOCSlist	<b>notify</b>
<sup>1)</sup> + denotes an included scenario. <sup>2)</sup> <variable> is shorthand for <variable> = true; !<variable> is shorthand for <variable> = false.			

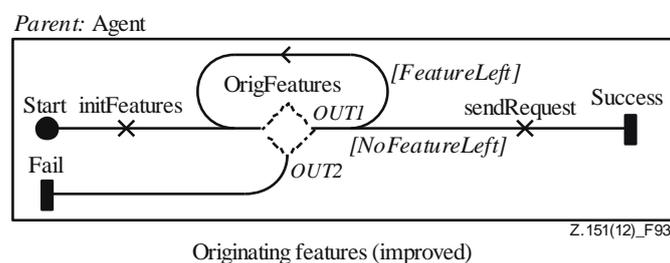
The last two scenario definitions are of particular interest because the more features are combined, the more likely it is that undesired interactions between these features manifest themselves in the UCM model. The penultimate scenario definition is not problematic, but the last one is. If the UCM path traversal attempts to run this scenario, the traversal will encounter two active plug-in maps for the TermFeatures stub, because the preconditions for

both the Teen Line plug-in map and the Originating Call Screening plug-in map are fulfilled. Both plug-in maps are therefore run in parallel. However, only the Teen Line feature fails and ends at the notify end point. The Originating Call Screening feature, however, succeeds and continues to the ring and ringing end points. This is an undesired interaction, because the active Teen Line feature has been circumvented and the scenario ends at both ring/ringing as well as notify.

There are different ways of resolving this conflict. One solution requires the following changes to the UCM model, as shown in Figure 93:

- Add two new Boolean variables: chkOCS and chkTL.
- Add responsibility `initFeatures` just before the `OrigFeatures` stub (initializes the new variables: `chkOCS = subOCS`; `chkTL = subTL`).
- Change the precondition for the OCS plug-in map of the `OrigFeatures` stub: `chkOCS`.
- Change the precondition for the Teen Line plug-in map of the `OrigFeatures` stub: `chkTL` and not `chkOCS`.
- Change the precondition for the Default plug-in map of the `OrigFeatures` stub: `not (chkOCS or chkTL)`.
- Add an OR-fork just after the `OrigFeatures` stub on the `OUT1` out-path.
- Loop back to the `OrigFeatures` stub if (`chkOCS or chkTL`) (`FeatureLeft` branch), continue otherwise (`NoFeatureLeft` branch).
- Add a variable assignment to the responsibility `checkOCS` on the Originating Call Screening plug-in map (`chkOCS = false`).
- Add a variable assignment to the responsibility `checkTime` on the Teen Line plug-in map (`chkTL = false`).

This solution gives priority to OCS over Teen Line (see preconditions of plug-in maps) because OCS does not require user interaction and it is not worth asking the originating user for a PIN if the call is blocked by OCS. This solution does not run the plug-in maps in parallel but one at the time. Each time the traversal reaches the `OrigFeatures` stub, a different feature is chosen, because a `chk<Feature>` variable was changed by a responsibility on the previously run plug-in map (see the variable assignments in the last two bullets).



**Figure 93 – Example: UCM model (improved)**

NOTE – All approaches for structuring scenario definitions in this clause could have specified preconditions and postconditions for the scenarios in addition to start point, end points and initializations but did not for simplicity of the example UCM model.

### 8.5.3 Initialization

*Initialization* specifies the initial value of a variable in a scenario definition (see Figure 92).

a) *Abstract grammar*

#### i) Attributes

- value (String): The initial value of a variable.

## ii) Relationships

- Contained by *ScenarioDef* (1): An *Initialization* is contained in one scenario definition.
- Association with *Variable* (1): An *Initialization* is for one variable.

## iii) Constraints

- The value shall be a Boolean literal as defined in clause 9.2.1, if the type of the *Variable* is Boolean.
- The value shall be an *Integer* literal (possibly preceded by the additive complement operator) as defined in clause 9.2.2, if the type of the *Variable* is Integer.
- The value shall be an enumeration literal from the values of the *EnumerationType* of the *Variable* as defined in clause 9.2.3, if the type of the *Variable* is Enumeration.

## b) Concrete grammar

An *Initialization* does not have a graphical visual representation. Optionally, an *Initialization* is visualized as part of its *ScenarioDef* using the textual syntax for *Initialization* as specified in Annex B.

## c) Semantics

*Initializations* are used by the traversal of a UCM path to set variables of the global data model in the URN specification (see clause 9.1) before traversing the UCM model based on a scenario definition.

## 8.5.4 Variable

*Variables* are part of the global data model in the URN specification (see clause 9.1). A variable may be one of several types and is initialized by the UCM traversal mechanism with *Initialization* values (see Figure 92).

## a) Abstract grammar

### i) Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 59).
- type (*DatatypeKind*): The type of the variable. Default value is Boolean.

### ii) Relationships

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *Variable* is contained in the UCM specification (see Figure 58).
- Association with *EnumerationType* (0..1): A *Variable* may be of Enumeration type.
- Uses *DatatypeKind* enumeration.

### iii) Constraints

- Inherits constraints from *UCMmodelElement*.
- Any two *Variables* cannot share the same name inside a URN specification.
- A *Variable* has one *EnumerationType* if and only if the type of the *Variable* is Enumeration.
- The name of a *Variable* shall be different from the following keywords: and, or, xor, not, mod, true, false, if, else, UCMFailures as well as all keywords for TURN as specified in Annex B.
- The name of a *Variable* shall not end with "\_pre".

b) *Concrete grammar*

A *Variable* does not have a graphical visual representation. Optionally, a *Variable* is visualized using the textual syntax for *Variable* as specified in Annex B.

**i) Relationships**

- Inherits relationships from *UCMmodelElement*.

c) *Semantics*

*Variable* is defined by its type and may be initialized according to an *Initialization* at the beginning of the traversal of a UCM path. The value of a variable may change during the traversal of a UCM path because of the expression of a *Responsibility* definition. The initial value of a *Variable* provided by its *Initialization* is accessible in data expressions (see clause 9.3) by using its name with the "\_pre" suffix. A variable has a data item of the same type, or it is "undefined".

### 8.5.5 EnumerationType

*EnumerationType* defines the valid values of an *Enumeration* (see Figure 92).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *UCMmodelElement* (see Figure 59).
- values (String): A comma-separated list of values specifies the valid choices for an *Enumeration* type.

**ii) Relationships**

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): An *EnumerationType* is contained in the UCM specification (see Figure 58).
- Association with *Variable* (0..\*): An *EnumerationType* may be used for variables.

**iii) Constraints**

- Inherits constraints from *UCMmodelElement*.
- values shall be a comma-separated list of enumeration literals as defined in clause 9.2.3.
- Each individual value in values shall be unique within values.
- Each individual value in values shall be different from all *Variable* names.
- Each individual value in values shall not end with "\_pre".
- Each individual value in values shall be different from the following keywords: and, or, xor, not, mod, true, false, if, else as well as all keywords for TURN as specified in Annex B.

b) *Concrete grammar*

An *EnumerationType* does not have a graphical visual representation. Optionally, an *EnumerationType* is visualized using the textual syntax for *EnumerationType* as specified in Annex B.

**i) Relationships**

- Inherits relationships from *UCMmodelElement*.

c) *Semantics*

*EnumerationType* specifies the choices of valid values for a variable of type *Enumeration*. An *Enumeration* consists of at least one String value. If more than one value is specified for

an Enumeration, a list of values separated by commas is used. For example, "Value1,Value2" specifies an Enumeration with two valid values.

### 8.5.6 DatatypeKind

A variable can be a Boolean, an Integer or an Enumeration (see Figure 92).

a) *Abstract grammar*

i) **Attributes**

– None (enumeration metaclass).

ii) **Relationships**

– Used by *Variable*.

iii) **Constraints**

– None.

b) *Concrete grammar*

None (enumeration metaclass). However, it influences the presentation of variables using the textual syntax (see clause 8.5.4).

c) *Semantics*

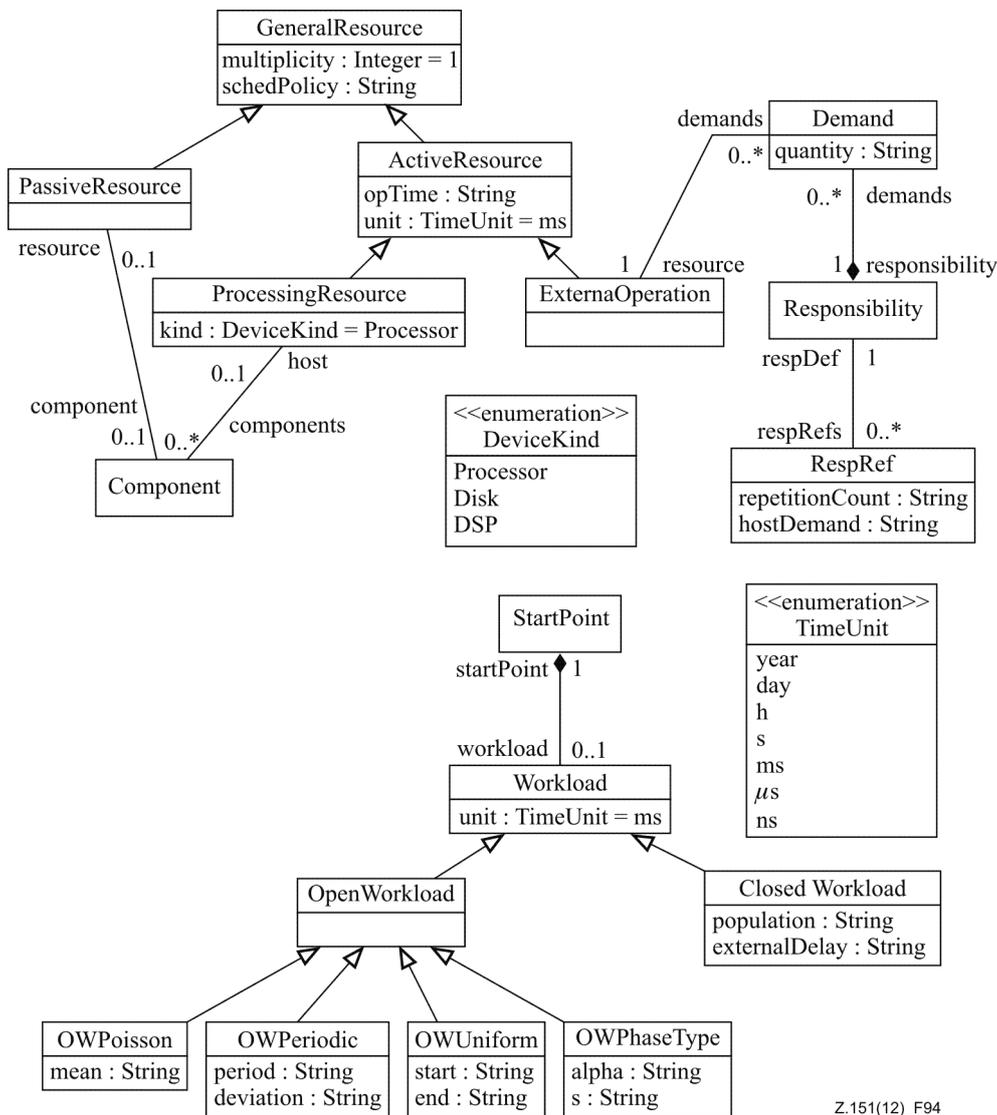
*DatatypeKind* defines the data type for a variable and therefore influences the valid values, valid expressions and valid uses a variable may have. All of these are defined for *Boolean*, *Integer* and *Enumeration* in more detail in clause 9 – Data language.

## 8.6 UCM performance annotations

UCM performance annotations enable the performance analysis of UCM models. Different kinds of resources can be defined so that potential deployments and usages of UCM components supporting scenarios can be analysed. A *Component* may be hosted on a *ProcessingResource* or may be considered as a *PassiveResource*. A *RespRef* may make a demand on the processing resource hosting its containing component. A *Responsibility* may also make explicit demands (e.g., service requests) on *ExternalOperations*. A *StartPoint* may contain a *Workload* describing the load density applied to a scenario. Workloads can be open with different arrival distributions, or closed with a fixed population. The time units used are specified in the workload (see Figure 94).

These annotations, together with others in metaclasses previously defined (*hostDemand* and *repetitionCount* in *RespRef*, *probability* in *NodeConnection*, and *probability* in *PluginBinding*), enable transformations of UCM models to models specified in languages suitable for performance analysis. Such transformations are, however, outside the scope of this Recommendation.

There are no specific concrete grammar metaclasses for the model elements defined in this clause.



Z.151(12)\_F94

**Figure 94 – Abstract grammar: UCM performance annotations**

### 8.6.1 Workload

*Workload* is a characterization of the load intensity applied to a scenario initiated at a *StartPoint*. A workload is open or closed and specifies a time unit (see Figure 94).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *UCMmodelElement* (see Figure 59).
- *unit* (*TimeUnit*): The unit of time used by the other attributes of the workload. Default value is ms (millisecond).

**ii) Relationships**

- Inherits relationships from *UCMmodelElement*.
- Contained by *StartPoint* (1): A *Workload* is contained in one start point.
- Uses *TimeUnit* enumeration.
- *Workload* is a superclass of *OpenWorkload* and *ClosedWorkload*.

**iii) Constraints**

- Inherits constraints from *UCMmodelElement*.

- All instances of *Workload* shall appear in one of its subclasses (that is, metaclass *Workload* is abstract).

*b) Concrete grammar*

A *Workload* does not have a visual representation.

**i) Relationships**

- Inherits relationships from *UCMmodelElement*.

*c) Semantics*

A *Workload* characterizes the requests for a scenario at a start point. A workload shall be either open (*OpenWorkload*) or closed (*ClosedWorkload*).

The time unit used in the definition of workload parameters (in *OpenWorkload* and *ClosedWorkload* subclasses) are specified by the unit attribute.

### 8.6.2 TimeUnit

The time unit used by a workload definition can be year (year), day (day), hour (h), second (s), millisecond (ms), microsecond ( $\mu$ s) or nanoseconds (ns) (see Figure 94).

*a) Abstract grammar*

**i) Attributes**

- None (enumeration metaclass).

**ii) Relationships**

- Used by *Workload* and *ActiveResource*.

**iii) Constraints**

- None.

*b) Concrete grammar*

None (enumeration metaclass).

*c) Semantics*

A time unit (used by a workload) can be one of the following:

- year: year (365 days)
- day: day (24 hours)
- h: hour (3600 seconds)
- s: second
- ms: millisecond (1/1000 second)
- $\mu$ s: microsecond (1/1000 millisecond)
- ns: nanosecond (1/1000 microsecond)

### 8.6.3 ClosedWorkload

A *ClosedWorkload* is a *Workload* with a fixed number of active users that cycle through the system (see Figure 94).

*a) Abstract grammar*

**i) Attributes**

- Inherits attributes from *Workload*.
- population (String): The number of active users in a closed workload.

- externalDelay (String): The mean think time of users between requests on the containing start point.

#### ii) Relationships

- Inherits relationships from *Workload*.

#### iii) Constraints

- Inherits constraints from *Workload*.
- The population shall be an Integer expression, as defined in clause 9.3.
- The population shall evaluate to a non-negative Integer value.
- The externalDelay shall be an Integer expression, as defined in clause 9.3.
- The externalDelay shall evaluate to a non-negative Integer value.

#### b) Concrete grammar

A *ClosedWorkload* does not have a visual representation.

#### i) Relationships

- Inherits relationships from *Workload*.

#### c) Semantics

A *ClosedWorkload* is a *Workload* that has a population attribute defining the finite number of active users of the containing scenario start point, as well as an external delay (externalDelay) defining the mean think time of each user between requests on that start point. The time unit of the external delay is specified by the workload's unit attribute.

### 8.6.4 OpenWorkload

An *OpenWorkload* is a *Workload* that represents streams of requests which arrive at a given rate in some predetermined pattern (see Figure 94).

#### a) Abstract grammar

##### i) Attributes

- Inherits attributes from *Workload*.

##### ii) Relationships

- Inherits relationships from *Workload*.
- *OpenWorkload* is a superclass of *OWPoisson*, *OWPeriodic*, *OWUniform* and *OWPhaseType*.

##### iii) Constraints

- Inherits constraints from *Workload*.
- All instances of *OpenWorkload* shall appear in one of its subclasses (that is, metaclass *OpenWorkload* is abstract).

#### b) Concrete grammar

An *OpenWorkload* does not have a visual representation.

#### i) Relationships

- Inherits relationships from *Workload*.

#### c) Semantics

An *OpenWorkload* is a *Workload* with an open arrival pattern defined by one of its subclasses. The arrival pattern represents streams of requests which arrive at the workload's start point.

### 8.6.5 OWPoisson

*OWPoisson* is an *OpenWorkload* with a Poisson arrival distribution (see Figure 94).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *OpenWorkload*.
- mean (String): The mean of the Poisson distribution is the value of the mean attribute divided by 1000.

**ii) Relationships**

- Inherits relationships from *OpenWorkload*.

**iii) Constraints**

- Inherits constraints from *OpenWorkload*.
- The mean shall be an Integer expression, as defined in clause 9.3.
- The mean shall evaluate to a positive Integer value.

b) *Concrete grammar*

An *OWPoisson* does not have a visual representation.

**i) Relationships**

- Inherits relationships from *Workload*.

c) *Semantics*

*OWPoisson* is an *OpenWorkload* with a mathematical Poisson arrival distribution. The mean of the distribution (often called  $\lambda$  in mathematics) is the mean attribute (a positive Integer) divided by 1000.

### 8.6.6 OWPeriodic

*OWPeriodic* is an *OpenWorkload* with a periodic arrival (see Figure 94).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *OpenWorkload*.
- period (String): The period of the periodic arrival.
- deviation (String): The maximal deviation of the periodic arrival.

**ii) Relationships**

- Inherits relationships from *OpenWorkload*.

**iii) Constraints**

- Inherits constraints from *OpenWorkload*.
- The period shall be an Integer expression, as defined in clause 9.3.
- The period shall evaluate to a positive Integer value.
- The deviation shall be an Integer expression, as defined in clause 9.3.
- The deviation shall evaluate to a non-negative Integer value.

b) *Concrete grammar*

An *OWPeriodic* does not have a visual representation.

**i) Relationships**

- Inherits relationships from *Workload*.

c) *Semantics*

*OWPeriodic* is an *OpenWorkload* with a periodic arrival characterized by a period and a maximal deviation. The time units of the period and deviation are specified by the workload's unit attribute.

### 8.6.7 OWUniform

*OWUniform* is an *OpenWorkload* with a uniform arrival distribution and a sampling interval (see Figure 94).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *OpenWorkload*.
- start (String): The start of the sampling interval.
- end (String): The end of the sampling interval.

**ii) Relationships**

- Inherits relationships from *OpenWorkload*.

**iii) Constraints**

- Inherits constraints from *OpenWorkload*.
- The start shall be an Integer expression, as defined in clause 9.3.
- The start shall evaluate to a non-negative Integer value.
- The end shall be an Integer expression, as defined in clause 9.3.
- The end shall evaluate to a positive Integer value greater than start's evaluated value.

b) *Concrete grammar*

An *OWUniform* does not have a visual representation.

**i) Relationships**

- Inherits relationships from *Workload*.

c) *Semantics*

*OWUniform* is an *OpenWorkload* with a mathematical (discrete) uniform arrival distribution. The time units of the start and end are specified by the workload's unit attribute.

### 8.6.8 OWPhaseType

*OWPhaseType* is an *OpenWorkload* with a phase-type arrival distribution (see Figure 94).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *OpenWorkload*.
- alpha (String): The probability row-vector of the phase-type arrival distribution. Each probability is expressed in thousandth.
- s (String): The subgenerator square matrix of the phase-type arrival distribution.

**ii) Relationships**

- Inherits relationships from *OpenWorkload*.

**iii) Constraints**

- Inherits constraints from *OpenWorkload*.

b) *Concrete grammar*

An *OWPhaseType* does not have a visual representation.

i) **Relationships**

- Inherits relationships from *Workload*.

c) *Semantics*

*OWPhaseType* is an *OpenWorkload* with a mathematical phase-type arrival distribution (that results from a system of one or more interrelated Poisson processes occurring in sequence, or phases). A phase-type distribution can be used to describe other types of distributions, which are special cases: exponential, Erlang, deterministic, Coaxian, hyper-exponential and hypo-exponential distributions.

The alpha attribute is vector of Integer values, where the values represent probabilities multiplied by 1000 and are separated by commas (e.g., "900, 100, 0").

The s attribute is a square matrix of Integer values, where each row is a vector between square brackets and each vector is separated by a comma (e.g., "[−3, 0, 0], [0, −4, 0], [0, 0, −5]").

The time units used in the s matrix are specified by the workload's unit attribute.

### 8.6.9 GeneralResource

*GeneralResource* is a UCM model element that represents a resource that can be used by responsibilities or that can be used to deploy components on (see Figure 94).

a) *Abstract grammar*

i) **Attributes**

- Inherits attributes from *UCMmodelElement* (see Figure 59).
- multiplicity (Nat): The number of available resources. Default value is 1.
- schedPolicy (String): The type of scheduling policy.

ii) **Relationships**

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *GeneralResource* is contained in the UCM specification (see Figure 58).
- *GeneralResource* is a superclass of *PassiveResource* and *ActiveResource*.

iii) **Constraints**

- Inherits constraints from *UCMmodelElement*.
- All instances of *GeneralResource* shall appear in one of its subclasses (that is, metaclass *GeneralResource* is abstract).

b) *Concrete grammar*

A *GeneralResource* does not have a visual representation.

i) **Relationships**

- Inherits relationships from *UCMmodelElement*.

c) *Semantics*

*GeneralResource* is an abstract class used to define attributes common to the other resources. The multiplicity represents the number of copies of the same resource. An optional schedPolicy attribute can be used to assign a specific scheduling policy to the resource, but its format is outside the scope of this Recommendation.

### 8.6.10 PassiveResource

*PassiveResource* is a *GeneralResource* that represents a resource that can be acquired and released (see Figure 94).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *GeneralResource*.

**ii) Relationships**

- Inherits relationships from *GeneralResource*.
- Association with *Component* (0..1): A *PassiveResource* may have one component definition.

**iii) Constraints**

- Inherits constraints from *GeneralResource*.

b) *Concrete grammar*

A *PassiveResource* does not have a visual representation.

**i) Relationships**

- Inherits relationships from *GeneralResource*.

c) *Semantics*

Passive resources are resources that do not have their own threads of control. Passive resources represent resources that shall be held but which do not perform operations. A *Component* associated to a passive resource represents that passive resource on UCM diagrams.

### 8.6.11 ActiveResource

*ActiveResource* is a *GeneralResource* that executes or processes its operations itself within the context of a performance model (see Figure 94).

a) *Abstract grammar*

**i) Attributes**

- Inherits attributes from *GeneralResource*.
- *opTime* (String): The time required by the *ActiveResource* to do one operation.
- *unit* (*TimeUnit*): The unit of time used by *opTime*. Default value is ms (millisecond).

**ii) Relationships**

- Inherits relationships from *GeneralResource*.
- *ActiveResource* is a superclass of *ProcessingResource* and *ExternalOperation*.
- Uses *TimeUnit* enumeration.

**iii) Constraints**

- Inherits constraints from *GeneralResource*.
- All instances of *ActiveResource* shall appear in one of its subclasses (that is, metaclass *ActiveResource* is abstract).
- The *opTime* shall be an Integer expression, as defined in clause 9.3.
- The *opTime* shall evaluate to a non-negative Integer value.

b) *Concrete grammar*

An *ActiveResource* does not have a visual representation.

**i) Relationships**

- Inherits relationships from *GeneralResource*.

**c) Semantics**

*ActiveResources* are resources that have their own thread of control. Active resources represent resources that perform operations. The *opTime* attribute describes the time needed by the resource to perform one operation, in the time unit specified.

### 8.6.12 ProcessingResource

*ProcessingResource* is an *ActiveResource* that represents a hardware processor (see Figure 94).

**a) Abstract grammar**

**i) Attributes**

- Inherits attributes from *ActiveResource*.
- *kind* (*DeviceKind*): The specific kind of hardware processing device represented by the resource. Default value is *Processor*.

**ii) Relationships**

- Inherits relationships from *ActiveResource*.
- Association with *Component* (0..\*): A *ProcessingResource* may have component definitions for which it acts as a host.
- Uses *DeviceKind* enumeration.

**iii) Constraints**

- Inherits constraints from *ActiveResource*.

**b) Concrete grammar**

A *ProcessingResource* does not have a visual representation.

**i) Relationships**

- Inherits relationships from *ActiveResource*.

**c) Semantics**

A *ProcessingResource* represents a hardware host for the software *Components* associated with it. The *kind* attribute defines the type of hardware being represented by the resource.

### 8.6.13 DeviceKind

A processing resource can be a *Processor*, a *Disk* or a digital signal processor (*DSP*) (see Figure 94).

**a) Abstract grammar**

**i) Attributes**

- None (enumeration metaclass).

**ii) Relationships**

- Used by *ProcessingResource*.

**iii) Constraints**

- None.

**b) Concrete grammar**

None (enumeration metaclass).

c) *Semantics*

*DeviceKind* is an enumerated type representing one of three kinds of hardware (processor, disk or DSP).

#### 8.6.14 ExternalOperation

An *ExternalOperation* is an *ActiveResource* that represents services provided by external devices which are not defined in the current model (see Figure 94).

a) *Abstract grammar*

**i) Attributes**

– Inherits attributes from *ActiveResource*.

**ii) Relationships**

– Inherits relationships from *ActiveResource*.

– Association with *Demand* (0..\*): An *ExternalOperation* may have demands made on it.

**iii) Constraints**

– Inherits constraints from *ActiveResource*.

b) *Concrete grammar*

An *ExternalOperation* does not have a visual representation.

**i) Relationships**

– Inherits relationships from *ActiveResource*.

c) *Semantics*

An *ExternalOperation* represents a service performed by a resource or set of resources defined outside of the scope of the current model. *ExternalOperations* are used to describe operations done by external services.

#### 8.6.15 Demand

A *Demand* describes an average service request to (or use of) an *ExternalOperation* performed by a *Responsibility* (see Figure 94).

a) *Abstract grammar*

**i) Attributes**

– quantity (String): The average number of requests to the *ExternalOperation* per use of the scenario is the value of the quantity attribute divided by 1000.

**ii) Relationships**

– Contained by *Responsibility* (1): A *Demand* is contained in one responsibility definition.

– Association with *ExternalOperation* (1): A *Demand* is for one external operation.

**iii) Constraints**

– The quantity shall be an Integer expression, as defined in clause 9.3.

– The quantity shall evaluate to a non-negative Integer value.

b) *Concrete grammar*

None.

c) *Semantics*

A *Demand* quantifies the average number of service requests of a *Responsibility* to an *ExternalOperation* per traversal of a scenario. A responsibility can have multiple demands to

external operations, in addition to its own `hostDemand` (on the processing resource that hosts the component that contains the responsibility).

## 8.7 UCM concrete grammar metaclasses

The only concrete grammar metaclasses specific to the UCM notation is the direction arrow. All other concrete grammar metaclasses that may be contained by some of the UCM abstract grammar metaclasses have been covered in previous clauses. Concrete condition, label, position, size, concrete style and comment have already been covered in clauses 6.2.3, 7.8.8, 7.8.10, 7.8.11, 7.8.12 and 7.8.13, respectively (see Figures 7, 52, 53, 54, 55 and 56, respectively). See Figure I.10 for a complete overview of the concrete grammar metaclasses for the UCM notation.



**Figure 95 – Concrete grammar: DirectionArrow metaclasses**

### 8.7.1 DirectionArrow

*DirectionArrow* visualizes the direction of causal flow of a path in a UCM diagram. They are useful for long paths or for paths whose visualization would be otherwise ambiguous in terms of direction (see Figure 95).

a) *Abstract grammar*

*DirectionArrow* has no abstract syntax.

i) **Attributes**

– Inherits attributes from *PathNode*.

ii) **Relationships**

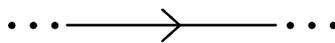
– Inherits relationships from *PathNode*.

iii) **Constraints**

– Inherits constraints from *PathNode*.

b) *Concrete grammar*

The symbol for *DirectionArrow* on a UCM path is defined as an open arrow-head ( $\rightarrow$ ) pointing towards the successor node connection (see Figure 96).



**Figure 96 – Symbol: UCM direction arrow**

i) **Relationships**

– Inherits relationships from *PathNode*.

ii) **Constraints**

– Inherits constraints from *PathNode*.

– A *DirectionArrow* is the source *PathNode* of exactly one *NodeConnection*.

– A *DirectionArrow* is the target *PathNode* of exactly one *NodeConnection*.

c) *Semantics*

*DirectionArrow* is a special case in that it is a concrete grammar metaclass that is specializing an abstract grammar metaclass. The abstract URN metamodel, therefore, ignores *PathNodes* of type *DirectionArrow* as if *DirectionArrows* do not exist, replacing the two *NodeConnections*

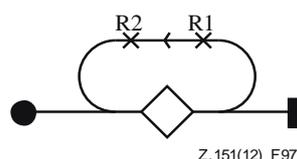
for a *DirectionArrow* (*PathNode* P1 → *DirectionArrow* → *PathNode* P2) with one *NodeConnection* (*PathNode* P1 → *PathNode* P2).

d) *Model*

None.

e) *Example*

The example in Figure 97 shows one stub with two in-paths and two out-paths. One pair of in-path and out-path forms a loop, first exiting the stub and then re-entering the stub. Although the direction of the loop is defined in the UCM model, the visualization of the loop is ambiguous if no direction arrow is used. With the direction arrow, however, it is possible to determine from the visualization that responsibility R2 follows responsibility R1, and not the opposite.



**Figure 97 – Example: UCM direction arrow**

## 9 Data language

URN supports simple data types and a data language sufficient to enable the evaluation of GRL strategies, the traversal of UCM models according to scenario definitions and UCM performance analysis. Though URN offers modellers a concrete syntax that is mostly graphical, the data language is textual and is based on a subset of the data language for SDL-2010 [ITU-T Z.104]. In order to accommodate modellers less experienced with SDL-2010, the concrete textual syntax also allows for operators from conventional programming languages (e.g., C and Java) to be used.

This clause is divided as follows:

- Clause 9.1: URN data model
- Clause 9.2: URN data types
- Clause 9.3: Grammar for expressions
- Clause 9.4: Grammar for actions

NOTE – Though the SDL-2010-like and programming language concrete syntaxes are supported by this grammar, it is recommended not to mix both styles to define the expressions and actions in a URN model.

### 9.1 URN data model

The data model of a URN specification is defined by the *Variables* the URN specification contains together with the annotations attached to URN model elements (e.g., performance annotations). In URN, variables are global. They are strongly typed and hence they can only contain values of their type (e.g., an Integer value cannot be assigned to a Boolean variable, or vice versa). Variables and annotations for which no value was provided are "undefined".

The primary intent of the URN data model is not to capture domain model data or complex data structures but to support GRL strategy evaluations, UCM scenario traversal and UCM performance annotations. It is hence simpler than most data languages found in programming languages and other specification languages.

## 9.2 URN data types

URN has three predefined basic data types, namely *Boolean*, *Integer* and *Enumeration* (see Figure 98). These represent a subset of the data types of [ITU-T Z.104], with support for a subset of the SDL-2010 syntax, operators and semantics.

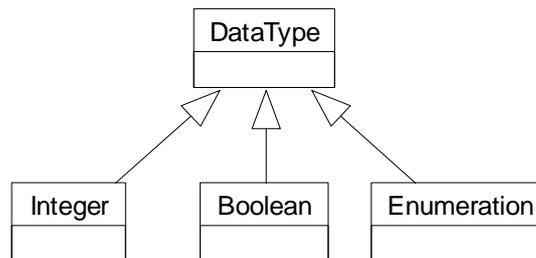


Figure 98 – URN data types

### 9.2.1 Boolean

The *Boolean* data type corresponds to a subset of the predefined Boolean sort in [ITU-T Z.104]. It is used to represent true and false values, which are the only two literals of this type:

```
<boolean literal> ::= true | false
```

Often a Boolean is used as the result of a comparison.

Table 13 lists the Boolean operators supported in URN, together with their SDL-2010 and alternative syntaxes, as well as signatures. The semantics is that of SDL-2010 Booleans.

Table 13 – Operators of the Boolean data type

<i>Operator</i>	<i>SDL-2010 syntax</i>	<i>Alternative syntax</i>	<i>Signature</i>
Equal	=	==	(Boolean, Boolean) → Boolean
Not Equal	/=	!=	(Boolean, Boolean) → Boolean
Negation	not	!	(Boolean) → Boolean
Conjunction	and	&&	(Boolean, Boolean) → Boolean
Disjunction	or		(Boolean, Boolean) → Boolean
Exclusive Disjunction	xor	^	(Boolean, Boolean) → Boolean
Logical Implication	=>	=>	(Boolean, Boolean) → Boolean

### 9.2.2 Integer

The *Integer* data type corresponds to a subset of the predefined Integer sort in [ITU-T Z.104]. It is used to represent mathematical integers with a decimal notation. The literals are non-empty sequences of decimal digits:

```
<integer literal> ::= <decimal digit>+
```

```
<decimal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Table 14 lists the Integer operators supported in URN, together with their SDL-2010 and alternative syntaxes, as well as signatures. The semantics is that of SDL-2010 Integers. However, unlike the division of SDL, division in URN does not generate any exception;<sup>1</sup> attempting to divide an Integer by zero results in an error (e.g., during the traversal of a UCM scenario, see clause 11.2).

<sup>1</sup> Though the handling of the exception raised by division is not defined in SDL-2010.

**Table 14 – Operators of the Integer data type**

<i>Operator</i>	<i>SDL-2010 syntax</i>	<i>Alternative syntax</i>	<i>Signature</i>
Equal	=	==	(Integer, Integer) → Boolean
Not Equal	/=	!=	(Integer, Integer) → Boolean
Greater Than	>	>	(Integer, Integer) → Boolean
Lower Than	<	<	(Integer, Integer) → Boolean
Greater or Equal to	>=	>=	(Integer, Integer) → Boolean
Lower or Equal to	<=	<=	(Integer, Integer) → Boolean
Addition	+	+	(Integer, Integer) → Integer
Subtraction	-	-	(Integer, Integer) → Integer
Multiplication	*	*	(Integer, Integer) → Integer
Division	/	/	(Integer, Integer) → Integer
Modulus	mod	%	(Integer, Integer) → Integer
Additive Complement	-	-	(Integer) → Integer

### 9.2.3 Enumeration

The *Enumeration* data type corresponds to a data type definition in [ITU-T Z.104] where only literals are listed (enumerating the elements of the sort). The literals are non-empty sequences of letters, decimal digits and underscores, but they are not allowed to start with a decimal digit or be solely composed of underscores. Each literal shall be unique within the enumeration type, and its name shall be different from operator keywords used in the grammars of clauses 9.3 and 9.4 (namely: and, or, xor, not, mod, true, false, if, else) as well as all keywords from TURN as specified in Annex B. A letter is a printable, alphabetical character from UCS [ITU-T T.55] and it can include accents or not. Letters are case sensitive.

```

<enumeration literal> ::= <identifier>
<identifier>          ::= {<letter> [<word>] | <underscore>+ <word>}
                        {<underscore>+ <word>}* <underscore>*
<word>                ::= {<letter> | <decimal digit>}+
<letter>              ::= (any character recommended in UCS [ITU-T T.55] with LETTER
                        as part of name of the character)
<underscore>         ::= _

```

Table 15 lists the Enumeration operators supported in URN, together with their SDL-2010 and alternative syntaxes, as well as signatures. The name of the *Enumeration* (EnumerationName) is the type of the enumeration's literals. Only equality and inequality operators are supported; there is no notion of ordering of the literals of the enumeration. Two enumeration literals compared for equality shall be of the same type (i.e., they are from the same *Enumeration* type).

**Table 15 – Operators of the Enumeration data type**

<i>Operator</i>	<i>SDL-2010 syntax</i>	<i>Alternative syntax</i>	<i>Signature</i>
Equal	=	==	(EnumerationName, EnumerationName) → Boolean
Not Equal	/=	!=	(EnumerationName, EnumerationName) → Boolean

### 9.3 Grammar for expressions

The following grammar defines the concrete syntax for URN data expression (<expression>). SDL-2010 operator precedence rules apply. As explained in clause 8.5.4, variable names are not allowed to end with "\_pre", so that this suffix can be used in expressions to refer to the initial value of the variable, as provided by a scenario definition.

```
<expression> ::= <implication>
<implication> ::= <disjunction> {<implies> <disjunction>}*
<disjunction> ::= <conjunction> {{<or> | <xor>} <conjunction>}*
<conjunction> ::= <comparison> {<and> <comparison>}*
<comparison> ::= <boolean unit> {{<equals> | <not equals>} <boolean unit>}*
<boolean unit> ::= <negation> | <relational expression> | <boolean literal>
<negation> ::= <not> <boolean unit>
<relational expression> ::=
    <additive expression> [ { <greater than>
        | <greater or equal to>
        | <lower than>
        | <lower or equal to> }
    <additive expression> ]
<additive expression> ::=
    <multiplicative expression> {{<addition> | <subtraction>}
    <multiplicative expression>}*
<multiplicative expression> ::=
    <unary expression> {
        {<multiplication> | <division> | <modulus>}
        <unary expression> }*
<unary expression> ::= [<addition> | <subtraction>]
    { <left parenthesis> <expression> <right parenthesis>
    | <integer literal>
    | <enumeration literal>
    | <variable> }

<equals> ::= = | ==
<not equals> ::= /= | !=
<and> ::= and | &&
<or> ::= or | ||
<xor> ::= xor | ^
<implies> ::= =>
<not> ::= not | !
<greater than> ::= >
<greater or equal to> ::= >=
<lower than> ::= <
<lower or equal to> ::= <=
<addition> ::= +
<subtraction> ::= -
<multiplication> ::= *
```

```

<division>          ::= /
<modulus>           ::= mod | %
<left parenthesis> ::= (
<right parenthesis> ::= )
<boolean literal>  ::= true | false
<integer literal>  ::= <decimal digit>+
<decimal digit>    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<enumeration literal> ::= <identifier>
<variable>         ::= <variable name> | <pre variable name>
<variable name>    ::= <identifier>
<pre variable name> ::= <variable name> <pre suffix>
<identifier>       ::= {<letter> [<word>] | <underscore>+ <word>}
                   {<underscore>+ <word>}* <underscore>*
<word>             ::= {<letter> | <decimal digit>}+
<letter>           ::= (any character recommended in UCS [ITU-T T.55] with LETTER
                   as part of the name of the character)
<underscore>      ::= _
<pre suffix>       ::= _pre

```

The type of an expression is computed using the signatures of the operators used. An expression between parentheses has the same type as the contained expression.

An `<identifier>` is an `<enumeration literal>` if it is one of the enumeration values of one of the *EnumerationTypes* defined in the URN specification. Its type is then the name of that *EnumerationType*.

An `<identifier>` is a `<variable name>` if it is the name of a *Variable* defined in the URN specification. Its type is then the type of that *Variable*.

An `<identifier>` that is neither an `<enumeration literal>` nor a `<variable name>` shall generate an error. A type mismatch shall also generate an error.

An expression that contains `<pre variable name>` shall only be used for postconditions of scenario definitions (see clause 8.5.2). Otherwise, an error shall be generated.

Of particular interests are the following types of expression, often used in String attributes of metaclasses:

- A *Boolean expression* is an `<expression>` whose computed type is *Boolean*.
- An *Integer expression* is an `<expression>` whose computed type is *Integer*.

## 9.4 Grammar for actions

URN supports a simple action language for modifying the values of variables. An action (`<action>`) is used by the expression attribute of a UCM *Responsibility* (see clause 8.2.4). Actions include simple assignments, compound statements and conditional statements. Actions are of a special type called Void, which is different from any data type, including user-defined enumerations. The different actions, together with their syntax and signatures, are defined in Table 16.

**Table 16 – Actions and their signatures**

<i>Operator</i>	<i>SDL-2010 syntax</i>	<i>Alternative syntax</i>	<i>Signature</i>
Boolean Assignment	<code>:=</code>	<code>=</code>	$(\text{Boolean}, \text{Boolean}) \rightarrow \text{Void}$
Integer Assignment	<code>:=</code>	<code>=</code>	$(\text{Integer}, \text{Integer}) \rightarrow \text{Void}$
Enumeration Assignment	<code>:=</code>	<code>=</code>	$(\text{EnumerationName}, \text{EnumerationName}) \rightarrow \text{Void}$
If Statement	<code>if</code>	<code>if</code>	$(\text{Boolean}, \text{Void}) \rightarrow \text{Void}$
If-Else Statement	<code>if else</code>	<code>if else</code>	$(\text{Boolean}, \text{Void}, \text{Void}) \rightarrow \text{Void}$
Compound Statement	<code>{ }</code>	<code>{ }</code>	$(\text{Void})^* \rightarrow \text{Void}$

The concrete syntax for actions is defined by the following grammar. It supplements the grammar for expressions defined in clause 9.3. The type of an action is computed according to the signatures defined in Table 16 and the rules from clause 9.3. A type mismatch shall generate an error.

```

<action> ::= <statement>*
<statement> ::= <assignment> | <compound statement> | <if statement>
<assignment> ::= <variable name> <assignment operator> <expression>
                <statement terminator>
<compound statement> ::= <left curly bracket> <statement>* <right curly bracket>
<if statement> ::= <if> <left parenthesis> <expression>
                <right parenthesis> <statement> [<else> <statement>]
<assignment operator> ::= = | :=
<if> ::= if
<else> ::= else
<statement terminator> ::= ;
<left curly bracket> ::= {
<right curly bracket> ::= }

```

For assignments, the value of a variable is replaced with the result of the evaluation of the <expression>. The types of <variable name> and of <expression> shall be the same; otherwise an error is generated. A value cannot be assigned to a *Variable* whose name ends with "`_pre`" (see clause 8.5.4), as such a variable name refers to the value of an *Initialization*.

Compound statements simply execute their inner statements in sequence.

An <if statement> executes its <statement> only when its Boolean <expression> evaluates to true. An <if-else statement> executes its first <statement> if its Boolean <expression> evaluates to true, and its second <statement> (after the else) if its Boolean <expression> evaluates to false.

## 9.5 Grammar for failures

URN supports a simple language for specifying failures or exceptions. A failure (<failure>) is used by the failure attribute of a UCM *FailurePoint* (see clause 8.2.17). A failure shall be the enumeration value (see clause 9.2.3) of the *EnumerationType* which has the reserved name UCMFailures and is defined in the URN specification. A failure list (<failure list>) is used by the failureList attribute of a UCM *StartPoint* (see clause 8.2.6). The failure list may be empty.

```

<failure list> ::= [<failure> {<comma> <failure>}*]
<failure> ::= <enumeration literal>
<comma> ::= ,

```

## 10 URN interchange format

Files describing URN models are expressed in XML. An XML schema definition (XSD) [W3C XSD1], specified in Annex A, defines the URN interchange format. It uses the UTF-8 encoding of UCS, which supports multiple natural languages, as recommended in [ITU-T T.55].

This schema is a serialization of the URN concrete syntax metamodel, which extends the abstract syntax metamodel. Concrete metaclasses are optional in the metamodel, and hence in the schema. Consequently, this format can be used to describe models based on the concrete syntax, or based solely on the abstract syntax.

The following rules were followed for the generation of the XSD schema in Annex A.

- **Enumeration class:** A *simpleType* element is declared for the enumeration class with the name attribute set to the class name. A restriction element is generated with base set to string. Each of the class attributes are appended to the restriction element as XSD enumeration elements with value set to the attribute name. They are presented first in the schema, in alphabetical order.
- **Normal metaclass:** A *complexType* element is declared for the metaclass with the name attribute set to the class name. The attributes and association roles are defined in a sequence element. They are also sorted in alphabetical order.
- **Data type:** URN and ITU-T Z.111 data types were converted to their XSD equivalent [W3C XSD2]: *xsd:nonNegativeInteger* for Nat, *xsd:boolean* for Boolean, *xsd:string* for String (Token) and *xsd:integer* for Integer. This enables some validation of the values by XML tools. The Boolean, Integer, Enumeration and DataType metaclasses are not included in the schema.
- **Attribute:** An element is declared for each class attribute. The element name is set to that of the attribute name of the metaclass. The XSD *minOccurs* and *maxOccurs* are left unspecified (*minOccurs* and *maxOccurs* default to 1). Default values are provided for enumeration types.
- **Identifiers:** The *id* attribute of *URNmodelElement* is of XSD type *ID*. An *ID* attribute of the same name was added to *NodeConnection* as the latter did not have an identifier while it is referenced by associations.
- **Association:** An element is declared for each navigable association owned by a class. The element name is set to that of the association role. The *minOccurs* and *maxOccurs* reflect the cardinality of the association (default to 1 if absent). *IDREF* is used as a type, with the target metaclass name provided as a comment. If the role is {ordered}, then *IDREFS* is used instead of *IDREF*.
- **Composition:** An element is declared for each of composition owned by a class. The element name is set to that of the association role. The *minOccurs* and *maxOccurs* reflect the cardinality of the association (default to 1 if absent). The name of the target metaclass is used as a type.
- **Inheritance:** An extension element is generated with the base attribute set to the base class name.
- **Root element:** *URNspec* is defined as the root element of the schema.
- **Constraints:** Static semantics constraints of the language are not enforced by the schema.

NOTE – In this XSD schema, when unspecified, the default values for elements of types *xsd:boolean*, *xsd:integer* and *xsd:string* are *false*, *0* and *''*, respectively.

## 11 URN analysis

URN models can be analysed in many ways. This clause focuses on two important techniques, namely GRL model evaluation and UCM scenario path traversal. As GRL models can be used for different

purposes, the goal-oriented modelling community has developed many analysis approaches. It is premature to standardize any of them, but GRL evaluation algorithms should be described according to different criteria and shall meet minimal requirements. Examples of evaluation algorithms are formalized and illustrated in Appendix II. The UCM path traversal mechanism is presented as a set of requirements that further define the dynamic semantics of UCM models. These requirements allow implementers to develop their own traversal algorithm and optimize or extend various aspects of it according to their needs. Two examples of a traversal algorithm are illustrated in Appendix III.

Although UCM models support a predefined set of performance annotations, quantitative performance analysis based on annotated UCMs is outside the scope of this Recommendation.

## 11.1 GRL model evaluation

During GRL model evaluation, for each *GRLLinkableElement* (i.e., GRL *IntentionalElement*, GRL *Indicator* and GRL *Actor*), three new *evaluation values* shall be created: one that is qualitative (qualitativeVal of type *QualitativeLabel*), one that is quantitative (quantitativeVal, an *Integer* value in the range [−100..100]) and one for the exceeds flag (exceedsVal, a Boolean value). Furthermore, a set of these three evaluation values is also created for the GRL model itself.

A GRL model is evaluated by assigning satisfaction values to a subset of the intentional elements and indicators (this is done with an *EvaluationStrategy*) and then propagating these values to the other intentional elements and indicators via the *ElementLinks* connecting them. Each *Evaluation* of an *EvaluationStrategy* shall see its values copied to its associated *GRLContainableElement* (i.e., *IntentionalElement* or *Indicator*): in the case of an *IntentionalElement*, evaluation is copied to quantitativeVal, qualitativeEvaluation is copied to qualitativeVal and exceeds is copied to exceedsVal. If the *Evaluation* is associated with an *Indicator*, then the realWorldValue and realWorldLabel of the *IndicatorEvaluation* of the *Evaluation* are first converted based on the *IndicatorConversion* of the *Evaluation*, before copying the resulting values to the quantitativeVal, qualitativeVal and exceeds evaluation attributes. The quantitativeVal, qualitativeVal and exceedsVal evaluation attributes of all other intentional elements, indicators, actors and the GRL model shall be initially set to 0, None and false, respectively.

From this initial context, goal models can be evaluated in many ways. Although no specific algorithm is given here, algorithms and tools for GRL model evaluation should consider the following non-exhaustive list of criteria when describing and formalizing evaluation algorithms.

- a) **Evaluation type:** Three general types of evaluations are considered:
  - *Quantitative evaluation:* uses the quantitativeContribution attribute of *Contribution*, the importanceQuantitative attribute of *GRLLinkableElements* and the new quantitativeVal attribute of *GRLContainableElements* initialized from the selected *EvaluationStrategy*.
  - *Qualitative evaluation:* uses the qualitative contribution attribute of *Contribution*, the importance attribute of *GRLLinkableElements* and the new attribute qualitativeVal of *GRLContainableElements* initialized from the selected *EvaluationStrategy*.
  - *Hybrid evaluation:* uses another combination of the above three categories of attributes (for contribution, importance and containable element evaluation value).
- b) **Propagation direction:** Three different propagation directions for the evaluation values through different GRL element links are considered:
  - *Forward propagation:* the *EvaluationStrategy* initializes some of the *IntentionalElements* and *Indicators* in the GRL model (source nodes, often leaves in the graph), and the evaluation values are propagated in a bottom-up way to higher-level intentional elements and indicators (targets) of the model. The results of selected alternatives can be analysed and conflicts detected.
  - *Backward propagation:* the *EvaluationStrategy* initializes some of the *IntentionalElements* and *Indicators* in the GRL model (target nodes, often roots in the

graph), and the evaluation values are propagated in a top-down way to lower-level intentional elements and indicators (sources) of the model. Such propagation is used to find a set of alternatives that, if satisfied, would lead to the initial values provided.

- *Mixed propagation*: a combination of the above where the *EvaluationStrategy* initializes some of the *IntentionalElements* and *Indicators* that are neither leaves nor roots. From these elements, forward propagation is used to compute evaluation values of higher-level intentional elements and indicators whereas backward propagation is used to compute evaluation values of lower-level intentional elements and indicators.

- c) **Overall GRL model satisfaction**: An algorithm may evaluate an evaluation level for the whole GRL model, or not.
- d) **Actor satisfaction**: An algorithm may evaluate actor evaluation levels, or not.
- e) **Exceeding expectations**: An algorithm may consider the *exceeds* attribute, or not. The *exceeds* attribute can be used to assess whether expectations are exceeded.
- f) **Automation**: An algorithm may be fully automated, or interactive (e.g., to resolve conflicts).
- g) **Cycles**: An algorithm may handle cycles in GRL models (completely or partially), or require models to be acyclic.
- h) **Conflicts**: An algorithm may determine that multiple contributions targeting the same intentional elements or indicators are conflicting, or not. In addition, if conflicts are detected, then there could be one or many categories of conflicts.
- i) **Strategy consistency**: An algorithm may allow inconsistent strategies, or not. An *EvaluationStrategy* is inconsistent if some of the initial evaluations it contains propagate into evaluations of intentional elements and indicators that are also initialized by the strategy, but with different values.
- j) **Evaluation overriding**: An algorithm may allow some evaluations defined as part of a strategy to be overridden during the propagation, or not.
- k) **Relation to UCM**: The results of the propagation may impact the values of UCM scenario variables, or not. In addition, updates to UCM scenario variables after path traversal may impact intentional element evaluations, or not.
- l) **Evaluation ordering for links**: GRL element links (decompositions, contributions and dependencies) may be evaluated in different orders. An algorithm should either specify that order or mention that there is no order.
- m) **Link evaluations**: An evaluation algorithm should provide functions to compute results of decomposition, contribution and dependency link usages.
- n) **Tolerance**: For contribution links, an algorithm may define a tolerance to help decide whether an intentional element becomes satisfied or just weakly satisfied (and respectively denied or just weakly denied) because of contributions.

Other criteria for which GRL could not easily offer support (e.g., probabilistic evaluations, separate values for satisfaction and denial, ranges of evaluation values instead of single values or optimistic and pessimistic evaluations) are outside the scope of this Recommendation.

GRL model evaluations are performed using the abstract syntax metaclasses, independently of the presence or absence of GRL diagrams. During model evaluations, however, the presentation of *IntentionalElementRefs* in GRL diagrams should be updated to reflect the current evaluation values (*quantitativeVal*, *qualitativeVal* and *exceedsVal*) of the referenced intentional element or indicator (see Figures 37 and 38). Similarly, the presentation of *ActorRef* and *CollapsedActorRef* in GRL diagrams should be updated to reflect the current evaluation values (*quantitativeVal*, *qualitativeVal* and *exceedsVal*) of the referenced actors (see Figures 32, 34 and 40).

## 11.2 UCM scenario path traversal

### 11.2.1 Overview

The path traversal (PT) mechanism is based on the abstract grammar metaclasses of the UCM notation. The path traversal mechanism traverses a UCM model by starting at the first start point as defined in a scenario definition by the modeller. The actual path to be traversed is determined by the initial, user-defined values of path variables and the changes to these values at responsibilities during the traversal. The path traversal mechanism moves from one path element to the next if path continuation criteria (PCC) are met. If more than one next path element meet the continuation criteria, all of these path elements are visited in parallel. Each UCM path element has specific criteria. The traversal ends when the last end point is reached. If the traversal stopped at a path element that is not an end point, a warning or error is issued. If the traversal cannot move from one currently visited path element to its next path element, the traversal continues with other path elements that are currently visited in parallel, if any exist. If none exist or all currently visited path elements cannot continue to their next path nodes, the traversal continues with the next start point in the ordered list of start points defined for the scenario definition. If the traversal is forcibly terminated, then the traversal is not allowed to continue for all path elements that are currently visited in parallel as well as all remaining start points and the traversal comes to a complete stop.

As a prerequisite for the path traversal, the start points, end points, initializations, preconditions and postconditions of all included scenario definitions shall be merged with the corresponding elements from the scenario definition itself as defined in the semantics of clause 8.5.2. The path traversal then operates on the merged scenarios as explained above. Initial values of the variables are accessible in postconditions by adding "\_pre" to a variable name. Since initializations of included scenarios may be overridden by the including scenario, the initial value of a variable refers to the initialization applied after the merging of included scenarios.

The path traversal mechanism as defined below assumes a sequential implementation of parallel paths. Furthermore, the choice of which parallel path to follow at any given time may be made at random since UCMs do not provide timing information sufficient enough for a more realistic simulation of parallel paths. If the path traversal mechanism encounters a non-deterministic choice point, a warning shall be issued. The traversal, however, may continue possibly by interacting with the modeller or by expanding multiple scenarios. If the path traversal mechanism continues – because of a failure or exception at a failure point – with a failure or abort start point on a map that is not in the map hierarchy at the time the traversal reached the failure point, the traversal continues and a warning may be issued. A warning is also issued if a failure or exception occurs at a failure point, but there is no failure or abort start point specified for the traversal to continue.

The current requirements for the path traversal mechanism (Table 17) cover all path elements and some component features. The path traversal mechanism is the basis for many advanced applications of UCMs. Most of these applications require additional capabilities. Scenario highlighting and animation can be done with the basic path traversal mechanism. The ability to associate path elements with sequence numbers indicating the order in which the path elements were traversed, however, makes repeated highlighting and animation more efficient. The generation of Message Sequence Charts requires the ability to deal with component information and a well-nestedness transformation/warning mechanism. The generation of performance models requires the ability to deal with arrival and device characteristics, device demands, data access modes and response-time requirements. Test case generation requires the ability to deal with information about controllable and observable activities. None of these additional capabilities, however, is currently a requirement for the path traversal mechanism.

### 11.2.2 Requirements for path traversal mechanism

The requirements for the path traversal mechanism use the terms *traversal root map*, *unconnected start point*, *abort scope* and *visit* which are shown in **bold** and underline in Table 17 and are defined in clause 3.

**Table 17 – Requirements for path traversal mechanism**

<b>ID</b>	<b>Requirement</b>
1	<i>Path traversal</i> (PT) shall start at the first start point of the scenario definition.
2	PT shall start with the initial values for path data variables as defined by the variable initializations of the scenario definition.
3	PT shall start with the special initial value "undefined" for path data variables not initialized by the variable initializations of the scenario definition.
4	PT shall evaluate an expression to "undefined" if any value within the expression evaluates to "undefined".
5	PT shall forcibly terminate the traversal if the result of an expression is "undefined".
6	PT shall forcibly terminate the traversal if the preconditions of the scenario definition are not fulfilled.
7	PT shall move in parallel from path element A to path elements B <sub>1</sub> , B <sub>2</sub> , ... and B <sub>N</sub> if <ol style="list-style-type: none"> <li>the traversal is currently visiting path element A, and</li> <li>there is a direct node connection from A to B<sub>1</sub>, from A to B<sub>2</sub>, ..., and from A to B<sub>N</sub>, and</li> <li>the <i>Path continuation criteria</i> (PCC) of A is fulfilled.</li> </ol>
8	PT shall continue at the next start point of the scenario definition if it cannot move to another path element from any of the currently visited path elements.
9	PT shall stop the traversal if <ol style="list-style-type: none"> <li>it cannot move to another path element from any of the currently visited path elements and</li> <li>there is no unused start point remaining in the scenario definition.</li> </ol>
10	PT shall issue a warning if <ol style="list-style-type: none"> <li>the traversal has stopped and</li> <li>the traversal is currently visiting at least one path element other than an end point.</li> </ol>
11	PT shall issue an error if the traversal has stopped and <ol style="list-style-type: none"> <li>at least one end point of the scenario definition has not been visited or</li> <li>at least one postcondition of the scenario definition is not fulfilled or</li> <li>a postcondition of at least one currently visited end point is not fulfilled.</li> </ol>
12	The PCC for a start point that is not a failure start point or abort start point shall be fulfilled if the precondition of the start point evaluates to true.
13	The PCC for an end point shall be fulfilled if the postcondition of the end point evaluates to true.
14	The PCC for a responsibility reference shall be always fulfilled.
15	Upon arrival at a responsibility reference, PT shall evaluate the expression for the repetition count of the responsibility reference.
16	After evaluating the repetition count of a responsibility reference, PT shall execute the expression of the responsibility definition of the responsibility reference as many times as specified by the repetition count.
17	PT shall execute the value assignment statements in the expression of a responsibility definition in the order defined by the modeller.
18	PT shall update the values of the path data variables immediately after executing one value assignment statement.

**Table 17 – Requirements for path traversal mechanism**

<i>ID</i>	<i>Requirement</i>
19	PT shall forcibly terminate the traversal if the execution of a value assignment statement results in a division by zero.
20	PT shall forcibly terminate the traversal if the evaluation of a condition results in a division by zero.
21	The PCC for an OR-fork shall be fulfilled if the PCC of exactly one branch of the OR-fork is fulfilled.
22	The PCC for a branch of an OR-fork shall be fulfilled if its condition evaluates to true.
23	The PCC shall forcibly terminate the traversal if a condition for a branch of an OR-fork is not specified.
24	The PCC for an OR-join shall be always fulfilled.
25	The PCC for an AND-fork shall be always fulfilled.
26	The PCC for an AND-join shall be fulfilled if the traversal is currently visiting the AND-join for all of its incoming paths.
27	The PCC for an empty point shall be always fulfilled.
28	The PCC for a connect shall be always fulfilled.
29	Upon arrival at a persistent waiting place, transient waiting place, persistent timer or transient timer along the waiting path WP, PT shall increase the number of arrivals along WP by 1 (the initial number of arrivals along WP is 0).
30	Upon arrival at a persistent waiting place or persistent timer along the trigger/release path TRP, PT shall increase the number of arrivals along TRP by 1 (the initial number of arrivals along TRP is 0).
31	Upon arrival at a transient waiting place or transient timer along the trigger/release path TRP, PT shall set the number of arrivals along TRP to 1, if the number of arrivals along the waiting path is greater than 0 (the initial number of arrivals along TRP is 0).
32	The PCC for a waiting place W shall be fulfilled if a) W's condition evaluates to true or b) the number of arrivals along the waiting path of W is at least one and the number of arrivals along the trigger/release path of W is at least one.
33	The PCC for a timer shall be fulfilled if the PCC of exactly one branch (i.e., either the PCC of its regular path or the PCC of its timeout path) is fulfilled.
34	The PCC for a regular path RP of a timer shall be fulfilled if the condition of RP evaluates to true.
35	The PCC of the regular path RP of a timer T shall be fulfilled if a) the condition of RP evaluates to false, and b) the condition of T's timeout path evaluates to false, and c) the number of arrivals along the waiting path of T is at least one and the number of arrivals along the trigger/release path of T is at least one.
36	The PCC for a timeout path of a timer T shall be fulfilled if the PCC of T's regular path is not fulfilled.
37	PT shall decrease NWP, the number of arrivals along the waiting path, by 1 when continuing past the waiting place or timer unless NWP is already 0.
38	PT shall decrease NTRP, the number of arrivals along the trigger/release path. by 1 when continuing past the waiting place or timer unless NTRP is already 0.
39	The PCC for a failure point FP shall be fulfilled if the condition of FP evaluates to true.
40	Upon arrival at a failure point FP with failure V and the condition of FP evaluating to false, PT shall traverse in parallel the failure and abort start points whose failure lists contain V.

**Table 17 – Requirements for path traversal mechanism**

<i>ID</i>	<i>Requirement</i>
41	Upon arrival at a failure start point or abort start point FAS, PT shall adjust the map hierarchy of the traversal by reducing it to the subset of the map hierarchy starting at the map of FAS.
42	Upon arrival at an abort start point A, PT shall stop the traversal of all path elements in the <b><u>abort scope</u></b> of A.
43	PT shall be deemed to be visiting a stub S if at least one path element on at least one plug-in map of S is being visited.
44	Upon arrival at a stub S, PT shall first traverse in parallel the plug-in maps of S before continuing with the traversal.
45	Upon arrival at a dynamic stub S, PT shall traverse in parallel the number of runtime instances of a plug-in map M of S as specified by the replication factor of the plug-in binding of M.
46	For each plug-in map runtime instance M of stub S, PT shall move in parallel from S to start points SP <sub>1</sub> , SP <sub>2</sub> , ... and SP <sub>N</sub> of plug-in map runtime instance M if <ul style="list-style-type: none"> <li>a) the traversal is currently visiting path element S, and</li> <li>b) the traversal has reached the stub via a direct node connection NC from path element A to S, and</li> <li>c) there is an in-binding of M from NC to SP<sub>1</sub>, from NC to SP<sub>2</sub>, ..., and from NC to SP<sub>N</sub>, and</li> <li>d) the PCC of the plug-in map runtime instance M is fulfilled.</li> </ul>
47	The PCC for the plug-in map runtime instance of a static stub shall be always fulfilled.
48	The PCC for a plug-in map runtime instance M of a dynamic stub shall be fulfilled if the precondition of the plug-in binding for M evaluates to true.
49	The PCC for an out-path O from the non-synchronizing stub S to a path element shall be fulfilled if <ul style="list-style-type: none"> <li>a) the traversal is visiting an end point E on a plug-in map runtime instance M of S, and</li> <li>b) an out-binding OB from E to O exists for M and OB belongs to the same plug-in binding used to arrive at M.</li> </ul>
50	The PCC for an out-path O from the synchronizing stub S to path element B shall be fulfilled if <ul style="list-style-type: none"> <li>a) the traversal has visited end points E<sub>1</sub>, E<sub>2</sub>, ... or E<sub>N</sub> on a plug-in map runtime instance M of S as often as specified by O's synchronization threshold during the same <b><u>visit</u></b>, and</li> <li>b) out-bindings OB<sub>1</sub> from E<sub>1</sub> to O, OB<sub>2</sub> from E<sub>2</sub> to O, ... and OB<sub>N</sub> from E<sub>N</sub> to O exist and OB<sub>1</sub>, OB<sub>2</sub>, ... and OB<sub>N</sub> belong to the same plug-in binding used to arrive at M.</li> </ul>
51	PT shall synchronize a synchronizing stub's plug-in map runtime instances, only if they belong to the same <b><u>visit</u></b> .
52	Once for each <b><u>visit</u></b> upon first arrival at a synchronizing stub S with the default synchronization threshold for an out-path O, PT shall set the synchronization threshold of O to the number of S's plug-in map runtime instances with preconditions evaluating to true.
53	PT shall ignore the arrival of plug-in map runtime instances at an out-path O of a synchronizing stub S during a <b><u>visit</u></b> , if the synchronization threshold of O has been reached for the <b><u>visit</u></b> .
54	Upon arrival at a synchronizing stub S with blocking enabled, PT shall allow an in-path of S to be traversed another time when all plug-in map runtime instances of S have been traversed.
55	When all plug-in map runtime instances of a synchronizing stub S have been traversed in the N <sup>th</sup> <b><u>visit</u></b> , PT shall treat an in-path of S as having been visited N times, if the in-path was visited less than N times.
56	Upon arrival at a singleton map M, PT shall traverse the only runtime instance of M that exists in the UCM model.
57	Upon arrival at an <b><u>unconnected start point</u></b> S of a non-singleton <b><u>traversal root map</u></b> M, PT shall traverse the N <sup>th</sup> runtime instance of M where N is the number of times S has been visited in the current traversal.

**Table 17 – Requirements for path traversal mechanism**

<i>ID</i>	<i>Requirement</i>
58	Upon arrival at a non-singleton plug-in map M of a non-synchronizing stub S, PT shall traverse a) a different runtime instance of M per different runtime instance of a stub and b) the same runtime instance of M for the same runtime instance of S.
59	Upon arrival at a non-singleton replicated plug-in map M of a non-synchronizing stub S, PT shall traverse a) a different set of replicated runtime instances of M per different runtime instance of a stub and b) the same set of replicated runtime instances of M for the same runtime instance of S.
60	Upon arrival at a non-singleton plug-in map M along an in-path of a synchronizing stub S, PT shall traverse a) a different runtime instance of M per different runtime instance of a stub, and b) the Nth runtime instance of M for this runtime instance of S during the Nth <b>visit</b> , and c) the same runtime instance of M for the same runtime instance of S in the same <b>visit</b> .
61	Upon arrival at a non-singleton replicated plug-in map M along an in-path of a synchronizing stub S, PT shall traverse a) a different set of replicated runtime instances of M per different runtime instance of a stub, and b) the Nth set of replicated runtime instances of M for this runtime instance of S during the Nth <b>visit</b> , and c) the same set of replicated runtime instances of M for the same runtime instance of S in the same <b>visit</b> .
62	Upon entering a protected component reference C along a path P, PT shall start the traversal of P when no other path is being traversed in any component reference of the component definition of C.
63	PT shall interleave path nodes of parallel branches that are bound to the same component reference C, if the component definition of C is of kind Object.
64	Upon arrival at a component reference C on a plug-in map runtime instance with a component plug-in binding to a component reference CP on the parent map runtime instance, PT shall use the component definition of CP as the component definition of C.
65	Upon arrival at a component reference C for which a plug-in binding is expected to be specified, PT shall issue a warning and continue with the traversal without replacing C.

## 12 Compliance statement

[ITU-T Z.150] lists each of the language requirements defined for the URN (FR and NFR). Table 18 recalls the requirements identified in Table 2 of [ITU-T Z.150] and provides the compliance statement of this Recommendation against these requirements.

Each language requirement possesses a unique identifier (ID) and his type. A language requirement is of type FR if it relates exclusively to functional requirements. A language requirement is of type NFR if it relates exclusively to non-functional requirements. A language requirement is of type URN if it is common to both functional and non-functional requirements. Language requirements are also defined as being essential (E), i.e., shall be supported, or desirable (D), i.e., should be supported. A language requirement is expressed as a capability the URN has. Table 18 lists all these attributes.

For each requirement in Table 18, compliance is established by listing the model elements defined in this Recommendation (or more precisely the clauses where they are defined) that satisfy the requirement. Brief additional explanations are provided, where needed, in the last column.

**Table 18 – Compliance statement of this Recommendation  
against ITU-T Z.150 language requirements**

<i>ID</i>	<i>ITU-T Z.150 language requirement</i>	<i>Type</i>	<i>E/D</i>	<i>ITU-T Z.151 clauses</i>	<i>Explanations</i>
1	Specify tentative and ill-defined requirements.	NFR	E	7.1.1, 7.3.1	Goal specifications, intentional elements
2	Specify refinement of goals and NFRs.	NFR	E	7.4.5	AND-type decompositions
3	Specify alternative refinement of goals and NFRs.	NFR	E	7.4.5	IOR-type and XOR-type decompositions
4	Specify alternative functional (operational) requirements.	NFR	E	7.4.5	IOR-type and XOR-type decompositions, means-end
5	Specify satisficeability of goals and NFRs.	NFR	E	7.5	Evaluation strategies
6	Support (qualitative) goals and NFRs that do not have clear metrics and measurements for their achievements.	NFR	E	7.3.1, 7.3.2	Intentional elements of type softgoal
7	Support quantitative goals and NFRs.	NFR	E	7.3.1, 7.3.2	Intentional elements of type goal
8	Specify trade-offs in goals and NFRs.	NFR	E	7.5	Evaluation strategies
9	Specify argumentation during modelling.	NFR	E	7.3.1, 7.3.2	Intentional elements of type belief
10	Specify business, organizational and system objectives.	NFR	E	7.2	GRL actors
11	Specify links between high-level objectives and lower-level specifications.	NFR	E	7.4, 6.1.3	Element links, URN links
12	Specify multiple stakeholders' requirements and interests.	NFR	E	7.2	GRL actors
13	Specify synergies and conflicts among goals and NFRs.	NFR	E	7.4	Element links
14	Support requirements priorities.	NFR	E	7.3.1	Importance attributes
15	Support negotiation for solving conflicting goals and NFRs.	NFR	E	7.5	Evaluation strategies
16	Support requirements evolution and changes.	NFR	E	6.1.2, 7.5	Identifiers, evaluation strategies (for regression)
17	Handle functional and non-functional requirements concurrently.	NFR	E	7.3.1, 7.3.2	Goals and softgoals
18	Specify selection criteria when choosing among alternative functional requirements.	NFR	E	7.5	Evaluation strategies
19	Support incremental commitments of requirements.	NFR	E	7.3.1, 7.5	Beliefs and evaluation strategies
20	Support requirements management during all development phases.	NFR	E	6.1.2	URNmodelElement with unique id attribute

**Table 18 – Compliance statement of this Recommendation  
against ITU-T Z.150 language requirements**

<i>ID</i>	<i>ITU-T Z.150 language requirement</i>	<i>Type</i>	<i>E/D</i>	<i>ITU-T Z.151 clauses</i>	<i>Explanations</i>
21	Have model elements that are identifiable and connectable to artefacts in external models.	NFR	E	6.1.2	URNmodelElement with unique id attribute
22	Support multiple levels of formality.	NFR	E	7.3.1, 6.1.3, 8.1.2	GRL Intentional elements, URN links, UCM model elements
23	Provide ease of use for customers and system users.	URN	E		Tool issue
24	Provide precise requirements for developers and testers.	NFR	E	6.1.3, 7.4.5	URN links, GRL decomposition
25	Support modular descriptions of goal and NFR models.	NFR	E	7.8.2, 8.2	GRL graphs and UCM maps
26	Support the reuse of goals, NFRs and knowledge in general.	NFR	D		Tool issue
27	Support performance indicators and mappings to satisfaction levels.	NFR	E	7.6	Indicators including evaluations of indicators and conversion methods
28	Support the mapping of input events and preconditions to output events and postconditions in various degrees of detail.	FR	E	8.2	UCM maps
29	Specify the set of input events at a scenario start point.	FR	E	8.2.6	Start point
30	Specify the set of output events at a scenario end point.	FR	E	8.2.8	End point
31	Specify preconditions at scenario start points.	FR	E	6.1.6, 8.2.6	Preconditions of start points
32	Specify postconditions at scenario end points.	FR	E	6.1.6, 8.2.8	Postconditions of end points
33	Specify input sources (human or machine).	FR	E	8.4	Components
34	Specify output sinks (human or machine).	FR	E	8.4	Components
35	Specify responsibilities and references to these responsibilities.	FR	E	8.2.4, 8.2.5	Responsibilities and responsibility references
36	Specify system operations as causal flows of responsibilities (paths).	FR	E	8.2.2, 8.2.3	Path nodes and node connections
37	Specify alternative paths.	FR	E	8.2.9	Or-forks
38	Specify common paths.	FR	E	8.2.10	Or-joins
39	Specify condition-based decision-making at branching points.	FR	E	6.1.6, 8.2.9	Conditions on Or-forks

**Table 18 – Compliance statement of this Recommendation  
against ITU-T Z.150 language requirements**

<i>ID</i>	<i>ITU-T Z.150 language requirement</i>	<i>Type</i>	<i>E/D</i>	<i>ITU-T Z.151 clauses</i>	<i>Explanations</i>
40	Define a data model and expression evaluator to express and evaluate conditions at branching points.	FR	E	9, 6.1.6, 8.2.9	URN data language, with use in conditions on Or-forks
41	Specify parallel or concurrent paths.	FR	E	8.2.11	And-forks
42	Specify synchronization of paths within a scenario.	FR	E	8.2.12	And-joins
43	Specify synchronization between paths from multiple scenarios.	FR	E	8.2.14, 8.2.16, 8.2.18	Waiting places, wait kinds and connects
44	Specify timed synchronization, with a timeout path.	FR	E	8.2.15, 8.2.16, 8.2.18	Timers, wait kinds and connects
45	Specify repetitive actions within a scenario.	FR	E	8.2.5	Responsibility reference with repetitionCount attribute
46	Support hierarchical decomposition of scenarios.	FR	E	8.3	Stubs and plug-ins
47	Specify subscenarios as scenarios.	FR	E	8.3	Stubs and plug-ins
48	Specify subscenario preconditions and postconditions.	FR	E	8.3.2, 6.1.6	Conditions on plug-in bindings
49	Specify scenario containers with multiple subscenarios.	FR	E	8.3.1	Dynamic stubs
50	Define a data model and expression evaluator to select subscenarios in dynamic containers.	FR	E	9, 6.1.6, 8.3.1	URN data language, with use in conditions on dynamic stubs
51	Group-related scenarios.	FR	E	8.2, 8.5.1	UCM maps and scenario groups
52	Extract individual scenarios from grouped scenarios.	FR	E	8.5.2, 11.2	Scenario definitions and UCM path traversal mechanism
53	Specify individual scenarios using a data model and initializations.	FR	E	8.5.3, 9	Initializations and URN data model
54	Express desirable feature interactions in scenarios.	FR	E	8.5.2, 11.2	Scenario definitions and successful UCM path traversal mechanism
55	Detect undesirable feature interactions in scenarios.	FR	E	11.2	UCM path traversal mechanism errors and warnings
56	Specify scenario cancellation situations with scope.	FR	E	8.2.6, 8.2.7, 8.2.17	Abort start points with abort scope
57	Specify scenarios describing recovery mechanisms.	FR	E	8.2.6, 8.2.7, 8.2.17	Failure points, failure start points and abort start points

**Table 18 – Compliance statement of this Recommendation  
against ITU-T Z.150 language requirements**

<i>ID</i>	<i>ITU-T Z.150 language requirement</i>	<i>Type</i>	<i>E/D</i>	<i>ITU-T Z.151 clauses</i>	<i>Explanations</i>
58	Specify qualitative time-dependent behaviour in scenarios.	FR	E	8.2.15	Timers
59	Specify timer types in time-dependent behaviour.	FR	E	8.2.15, 8.2.16	Timers and wait kinds
60	Specify components and references to these components.	FR	E	8.4.1, 8.4.4	Components and component references
61	Specify scenarios without reference to components.	FR	E	8.2.2	Path nodes do not need to be bound to components
62	Specify scenarios where scenario elements are allocated to components.	FR	E	8.2.2, 8.4.1, 8.4.4	Path nodes may be bound to components
63	Specify abstract components and COTS.	FR	E	8.4.2, 8.4.5	Component types and component bindings
64	Specify dynamic entities.	FR	E	8.4.3, 8.4.5	Component kinds and component bindings
65	Specify system boundaries.	FR	E	8.4.1, 8.4.3	Components of kind other than Actor
66	Specify the behaviour of the system's environment.	FR	E	8.2.2, 8.4.1, 8.4.4	Path nodes not bound to components
67	Specify actors external to the system.	FR	E	8.4.1, 8.4.3	Components of kind Actor
68	Support backward traceability from URN to source documents.	URN	E	6.1.2, 6.1.4	Unique model element identifier, metadata
69	Support forward traceability from URN to the other models used in the development process.	URN	E	6.1.2, 6.1.4	Unique model element identifier, metadata
70	Support facilities to connect URN elements to external requirements objects.	URN	E	6.1.2, 6.1.4	Unique model element identifier, metadata
71	Enable transformations to elements of other languages in the ITU-T family of languages and of UML.	URN	D	11.2	UCM path traversal mechanism
72	Support traceability between operational aspects of goal/NFR models and responsibilities/scenarios in scenario models.	URN	E	6.1.3	URN links
73	Support traceability between performance constraints in NFR models and responsibilities/scenarios/response-time requirements in scenario models.	URN	E	6.1.3	URN links
74	Support the testing of requirements.	URN	E	8.5	Scenario definitions
75	Support testing based on requirements.	FR	E	8.5	Scenario definitions
76	Support the evaluation of the satisfaction of goals and NFRs.	NFR	E	7.5, Appendix II	Evaluation strategies, GRL evaluation algorithms

**Table 18 – Compliance statement of this Recommendation  
against ITU-T Z.150 language requirements**

<i>ID</i>	<i>ITU-T Z.150 language requirement</i>	<i>Type</i>	<i>E/D</i>	<i>ITU-T Z.151 clauses</i>	<i>Explanations</i>
77	Enable preliminary analysis of performance properties.	URN	E	8.6	UCM performance annotations
78	Attach performance/workload annotations to scenario elements.	FR	E	8.6	UCM performance annotations
79	Specify the environment's processing capacity, network delays and services provided.	FR	E	8.6	UCM performance annotations
80	Specify response times in terms of target fragments of scenarios.	FR	E	8.5.2, 8.6	Scenario definitions, UCM performance annotations
81	Specify identifiers for the model elements.	URN	E	6.1.2	URNmodelElement with unique id attribute
82	Specify document versions.	URN	E	6.2.1	specVersion
83	Support textual annotations traceable to graphical elements.	URN	E	7.8.13	Comments
84	Support a graphical representation of requirements.	URN	E	6.2, 7.8, 8.7	Concrete grammar metaclasses and graphical concrete syntax
85	Support a tool-oriented interchange format.	URN	E	10	URN interchange format
86	Support textual annotations displayable on conventional media.	URN	E	7.8.13	Comments
87	Support the grouping of any elements in a model.	URN	E	6.1.5	Concern
88	Support annotating any model element with name-value pairs.	URN	E	6.1.4	Metadata
89	Support linking any pair of model elements.	URN	E	6.1.3	URN links

### 13 Tool compliance

This clause defines the compliance for tools that claim to support the User Requirements Notation and which therefore should be capable of creating, editing, presenting and analysing valid URN specifications. The validity of a URN specification is defined as in clause 5.2.1.

#### 13.1 Definitions of valid tools

##### 13.1.1 Compliant URN tool

A tool that detects non-compliance of a description with this Recommendation. If the tool handles a superset notation, it is allowed to categorize non-compliance as a warning rather than a failure.

### **13.1.2 Valid URN tool**

A compliant URN tool that supports the abstract grammar defined in this Recommendation and fulfils the requirements of the GRL model evaluation and of the UCM path traversal mechanism defined in clause 11.

### **13.1.3 Valid graphical URN tool**

A valid URN tool that also supports the concrete grammar defined in this Recommendation.

### **13.1.4 Compliant GRL tool**

A tool that detects non-compliance of a GRL description with this Recommendation. If the tool handles a superset notation, it is allowed to categorize non-compliance as a warning rather than a failure.

### **13.1.5 Valid GRL tool**

A compliant GRL tool that supports the GRL abstract grammar defined in this Recommendation and fulfils the requirements of the GRL model evaluation defined in clause 11.1.

### **13.1.6 Valid graphical GRL tool**

A valid GRL tool that also supports the GRL concrete grammar defined in this Recommendation.

### **13.1.7 Compliant UCM tool**

A tool that detects non-compliance of a UCM description with this Recommendation. If the tool handles a superset notation, it is allowed to categorize non-compliance as a warning rather than a failure.

### **13.1.8 Valid UCM tool**

A compliant UCM tool that supports the UCM abstract grammar defined in this Recommendation and fulfils the requirements of the UCM path traversal mechanism defined in clause 11.2.

### **13.1.9 Valid graphical UCM tool**

A valid UCM tool that also supports the UCM concrete grammar defined in this Recommendation.

## **13.2 Conformance**

A conformance statement clearly identifying the language features and requirements not supported should accompany any tool that handles a subset of this Recommendation. If no conformance statement is provided, it shall be assumed that the tool is a valid graphical URN tool. It is therefore preferable to supply a conformance statement; otherwise, any unsupported feature allows the tool to be rejected as invalid.

## Annex A

### URN interchange format: XML schema

(This annex forms an integral part of this Recommendation.)

The following XML schema defines the URN interchange format. It is explained in clause 10.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!--
== XML Schema for the User Requirements Notation (Recommendation ITU-T Z.151)
== Version: 20120902
-->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" attributeFormDefault="unqualified"
            elementFormDefault="qualified">

  <!-- ===== -->
  <!-- ===== Root Element ===== -->
  <!-- ===== -->
  <xsd:element name="URNspec" type="URNspec"/>
  <!-- ===== -->
  <!-- ===== Simple Type Definitions ===== -->
  <!-- ===== -->
  <!-- ~~~~~ -->
  <!-- ComponentKind -->
  <!-- ~~~~~ -->
  <xsd:simpleType name="ComponentKind">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Team"/>
      <xsd:enumeration value="Object"/>
      <xsd:enumeration value="Process"/>
      <xsd:enumeration value="Agent"/>
      <xsd:enumeration value="Actor"/>
    </xsd:restriction>
  </xsd:simpleType>
  <!-- ~~~~~ -->
  <!-- ContributionType -->
  <!-- ~~~~~ -->
  <xsd:simpleType name="ContributionType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Make"/>
      <xsd:enumeration value="Help"/>
      <xsd:enumeration value="SomePositive"/>
      <xsd:enumeration value="Unknown"/>
      <xsd:enumeration value="SomeNegative"/>
      <xsd:enumeration value="Hurt"/>
      <xsd:enumeration value="Break"/>
    </xsd:restriction>
  </xsd:simpleType>
  <!-- ~~~~~ -->
  <!-- DatatypeKind -->
  <!-- ~~~~~ -->
  <xsd:simpleType name="DatatypeKind">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Boolean"/>
      <xsd:enumeration value="Integer"/>
      <xsd:enumeration value="Enumeration"/>
    </xsd:restriction>
  </xsd:simpleType>
  <!-- ~~~~~ -->
  <!-- DecompositionType -->
  <!-- ~~~~~ -->
  <xsd:simpleType name="DecompositionType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="AND"/>
      <xsd:enumeration value="XOR"/>
    </xsd:restriction>
  </xsd:simpleType>

```

```

    <xsd:enumeration value="IOR"/>
  </xsd:restriction>
</xsd:simpleType>
<!-- ~~~~~ -->
<!-- DeviceKind -->
<!-- ~~~~~ -->
<xsd:simpleType name="DeviceKind">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Processor"/>
    <xsd:enumeration value="Disk"/>
    <xsd:enumeration value="DSP"/>
  </xsd:restriction>
</xsd:simpleType>
<!-- ~~~~~ -->
<!-- FailureKind -->
<!-- ~~~~~ -->
<xsd:simpleType name="FailureKind">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Failure"/>
    <xsd:enumeration value="Abort"/>
    <xsd:enumeration value="None"/>
  </xsd:restriction>
</xsd:simpleType>
<!-- ~~~~~ -->
<!-- ImportanceType -->
<!-- ~~~~~ -->
<xsd:simpleType name="ImportanceType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="High"/>
    <xsd:enumeration value="Medium"/>
    <xsd:enumeration value="Low"/>
    <xsd:enumeration value="None"/>
  </xsd:restriction>
</xsd:simpleType>
<!-- ~~~~~ -->
<!-- IntentionalElementType -->
<!-- ~~~~~ -->
<xsd:simpleType name="IntentionalElementType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Softgoal"/>
    <xsd:enumeration value="Goal"/>
    <xsd:enumeration value="Task"/>
    <xsd:enumeration value="Resource"/>
    <xsd:enumeration value="Belief"/>
  </xsd:restriction>
</xsd:simpleType>
<!-- ~~~~~ -->
<!-- QualitativeLabel -->
<!-- ~~~~~ -->
<xsd:simpleType name="QualitativeLabel">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Denied"/>
    <xsd:enumeration value="WeaklyDenied"/>
    <xsd:enumeration value="WeaklySatisfied"/>
    <xsd:enumeration value="Satisfied"/>
    <xsd:enumeration value="Conflict"/>
    <xsd:enumeration value="Unknown"/>
    <xsd:enumeration value="None"/>
  </xsd:restriction>
</xsd:simpleType>
<!-- ~~~~~ -->
<!-- TimeUnit -->
<!-- ~~~~~ -->
<xsd:simpleType name="TimeUnit">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="year"/>
    <xsd:enumeration value="day"/>
    <xsd:enumeration value="h"/>
    <xsd:enumeration value="s"/>
    <xsd:enumeration value="ms"/>
  </xsd:restriction>
</xsd:simpleType>

```

```

    <xsd:enumeration value="us"/>
    <xsd:enumeration value="ns"/>
  </xsd:restriction>
</xsd:simpleType>
<!-- ~~~~~ -->
<!-- WaitKind -->
<!-- ~~~~~ -->
<xsd:simpleType name="WaitKind">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Transient"/>
    <xsd:enumeration value="Persistent"/>
  </xsd:restriction>
</xsd:simpleType>

<!-- ===== -->
<!-- ===== Complex Type Definitions ===== -->
<!-- ===== -->
<!-- ~~~~~ -->
<!-- ActiveResource -->
<!-- ~~~~~ -->
<xsd:complexType name="ActiveResource">
  <xsd:complexContent>
    <xsd:extension base="GeneralResource">
      <xsd:sequence>
        <xsd:element name="opTime" type="xsd:string"/>
        <xsd:element default="ms" name="unit" type="TimeUnit"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Actor -->
<!-- ~~~~~ -->
<xsd:complexType name="Actor">
  <xsd:complexContent>
    <xsd:extension base="GRLlinkableElement">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="collapsedRefs" type="xsd:IDREF"/>
          <!-- CollapsedActorRef -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="actorRefs" type="xsd:IDREF"/> <!-- ActorRef -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="elems" type="xsd:IDREF"/>
          <!-- GRLContainableElement -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ActorRef -->
<!-- ~~~~~ -->
<xsd:complexType name="ActorRef">
  <xsd:complexContent>
    <xsd:extension base="GRLmodelElement">
      <xsd:sequence>
        <xsd:element name="label" type="Label"/>
        <xsd:element name="actorDef" type="xsd:IDREF"/> <!-- Actor -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="nodes" type="xsd:IDREF"/> <!-- GRLNode -->
        <xsd:element name="pos" type="Position"/>
        <xsd:element name="size" type="Size"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- AndFork -->
<!-- ~~~~~ -->
<xsd:complexType name="AndFork">
  <xsd:complexContent>
    <xsd:extension base="PathNode"/>
  </xsd:complexContent>
</xsd:complexType>

```

```

<!-- ~~~~~ -->
<!-- AndJoin -->
<!-- ~~~~~ -->
<xsd:complexType name="AndJoin">
  <xsd:complexContent>
    <xsd:extension base="PathNode"/>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ClosedWorkload -->
<!-- ~~~~~ -->
<xsd:complexType name="ClosedWorkload">
  <xsd:complexContent>
    <xsd:extension base="Workload">
      <xsd:sequence>
        <xsd:element name="population" type="xsd:string"/>
        <xsd:element name="externalDelay" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- CollapsedActorRef -->
<!-- ~~~~~ -->
<xsd:complexType name="CollapsedActorRef">
  <xsd:complexContent>
    <xsd:extension base="GRLNode">
      <xsd:sequence>
        <xsd:element name="actor" type="xsd:IDREF"/> <!-- Actor -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Comment -->
<!-- ~~~~~ -->
<xsd:complexType name="Comment">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string"/>
    <xsd:element name="x" type="xsd:integer"/>
    <xsd:element name="y" type="xsd:integer"/>
    <xsd:element name="width" type="xsd:integer"/>
    <xsd:element name="height" type="xsd:integer"/>
    <xsd:element name="fillColor" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Component -->
<!-- ~~~~~ -->
<xsd:complexType name="Component">
  <xsd:complexContent>
    <xsd:extension base="UCMmodelElement">
      <xsd:sequence>
        <xsd:element name="kind" type="ComponentKind"/>
        <xsd:element name="protected" type="xsd:boolean"/>
        <xsd:element name="context" type="xsd:boolean"/>
        <xsd:element minOccurs="0" name="type" type="xsd:IDREF"/> <!-- ComponentType -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="includedComponents" type="xsd:IDREF"/>
          <!-- Component -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="includingComponents" type="xsd:IDREF"/>
          <!-- Component -->
        <xsd:element minOccurs="0" name="host" type="xsd:IDREF"/> <!-- ProcessingResource -->
        <xsd:element minOccurs="0" name="resource" type="xsd:IDREF"/> <!-- PassiveResource -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="compRefs" type="xsd:IDREF"/>
          <!-- ComponentRef -->
        <xsd:element minOccurs="0" name="style" type="ConcreteStyle"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

```

```

<!-- ~~~~~ -->
<!-- ComponentBinding -->
<!-- ~~~~~ -->
<xsd:complexType name="ComponentBinding">
  <xsd:sequence>
    <xsd:element name="id" type="xsd:ID" /> <!-- ADDED because ComponentBinding is not a URNmodelElement (no ID) -->
    <xsd:element name="parentComponent" type="xsd:IDREF"/> <!-- ComponentRef -->
    <xsd:element name="pluginComponent" type="xsd:IDREF"/> <!-- ComponentRef -->
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ComponentRef -->
<!-- ~~~~~ -->
<xsd:complexType name="ComponentRef">
  <xsd:complexContent>
    <xsd:extension base="UCMmodelElement">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="parentBindings" type="xsd:IDREF"/>
          <!-- ComponentBinding -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="pluginBindings" type="xsd:IDREF"/>
          <!-- ComponentBinding -->
        <xsd:element name="compDef" type="xsd:IDREF"/> <!-- Component -->
        <xsd:element minOccurs="0" name="label" type="Label"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="children" type="xsd:IDREF"/> <!-- ComponentRef -->
        <xsd:element minOccurs="0" name="parent" type="xsd:IDREF"/> <!-- ComponentRef -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="nodes" type="xsd:IDREF"/> <!-- PathNode -->
        <xsd:element minOccurs="0" name="pos" type="Position"/>
        <xsd:element minOccurs="0" name="size" type="Size"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ComponentType -->
<!-- ~~~~~ -->
<xsd:complexType name="ComponentType">
  <xsd:complexContent>
    <xsd:extension base="UCMmodelElement">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="instances" type="xsd:IDREF"/> <!-- Component -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Concern -->
<!-- ~~~~~ -->
<xsd:complexType name="Concern">
  <xsd:complexContent>
    <xsd:extension base="URNmodelElement">
      <xsd:sequence>
        <xsd:element minOccurs="0" name="condition" type="Condition"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="elements" type="xsd:IDREF"/>
          <!-- URNmodelElement -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ConcreteCondition -->
<!-- ~~~~~ -->
<xsd:complexType name="ConcreteCondition">
  <xsd:sequence>
    <xsd:element name="label" type="xsd:string"/>
    <xsd:element name="description" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ConcreteGRLspec -->
<!-- ~~~~~ -->

```

```

<xsd:complexType name="ConcreteGRLspec">
  <xsd:sequence>
    <xsd:element name="showAsMeansEnd" type="xsd:boolean"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ConcreteStrategy -->
<!-- ~~~~~ -->
<xsd:complexType name="ConcreteStrategy">
  <xsd:sequence>
    <xsd:element name="author" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ConcreteStyle -->
<!-- ~~~~~ -->
<xsd:complexType name="ConcreteStyle">
  <xsd:sequence>
    <xsd:element name="lineColor" type="xsd:string"/>
    <xsd:element name="fillColor" type="xsd:string"/>
    <xsd:element name="filled" type="xsd:boolean"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ConcreteURNspec -->
<!-- ~~~~~ -->
<xsd:complexType name="ConcreteURNspec">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string"/>
    <xsd:element name="author" type="xsd:string"/>
    <xsd:element name="created" type="xsd:string"/>
    <xsd:element name="modified" type="xsd:string"/>
    <xsd:element name="specVersion" type="xsd:string"/>
    <xsd:element name="urnVersion" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Condition -->
<!-- ~~~~~ -->
<xsd:complexType name="Condition">
  <xsd:sequence>
    <xsd:element name="expression" type="xsd:string"/>
    <xsd:element minOccurs="0" name="desc" type="ConcreteCondition"/>
    <xsd:element minOccurs="0" name="label" type="Label"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Connect -->
<!-- ~~~~~ -->
<xsd:complexType name="Connect">
  <xsd:complexContent>
    <xsd:extension base="PathNode"/>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Contribution -->
<!-- ~~~~~ -->
<xsd:complexType name="Contribution">
  <xsd:complexContent>
    <xsd:extension base="ElementLink">
      <xsd:sequence>
        <xsd:element default="Unknown" name="contribution" type="ContributionType"/>
        <xsd:element name="quantitativeContribution" type="xsd:integer"/>
        <xsd:element name="correlation" type="xsd:boolean"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ContributionChange -->

```

```

<!-- ~~~~~ -->
<xsd:complexType name="ContributionChange">
  <xsd:sequence>
    <xsd:element default="Unknown" name="newContribution" type="ContributionType"/>
    <xsd:element name="newQuantitativeContribution" type="xsd:integer"/>
    <xsd:element name="contribution" type="xsd:IDREF"/> <!-- Contribution -->
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ContributionContext -->
<!-- ~~~~~ -->
<xsd:complexType name="ContributionContext">
  <xsd:complexContent>
    <xsd:extension base="GRLmodelElement">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="changes" type="ContributionChange"/>
        <xsd:element maxOccurs="unbounded" name="groups" type="xsd:IDREF"/>
          <!-- ContributionContextGroup -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="parentContexts" type="xsd:IDREF"/>
          <!-- ContributionContext -->
        <xsd:element minOccurs="0" name="includedContexts" type="xsd:IDREFS"/>
          <!-- ContributionContext {ordered} -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ContributionContextGroup -->
<!-- ~~~~~ -->
<xsd:complexType name="ContributionContextGroup">
  <xsd:complexContent>
    <xsd:extension base="GRLmodelElement">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="contributes" type="xsd:IDREF"/>
          <!-- ContributionContext -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Decomposition -->
<!-- ~~~~~ -->
<xsd:complexType name="Decomposition">
  <xsd:complexContent>
    <xsd:extension base="ElementLink"/>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Demand -->
<!-- ~~~~~ -->
<xsd:complexType name="Demand">
  <xsd:sequence>
    <xsd:element name="quantity" type="xsd:string"/>
    <xsd:element name="resource" type="xsd:IDREF"/> <!-- ExternalOperation -->
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Dependency -->
<!-- ~~~~~ -->
<xsd:complexType name="Dependency">
  <xsd:complexContent>
    <xsd:extension base="ElementLink"/>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Description -->
<!-- ~~~~~ -->
<xsd:complexType name="Description">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string"/>

```

```

</xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- DirectionArrow -->
<!-- ~~~~~ -->
<xsd:complexType name="DirectionArrow">
  <xsd:complexContent>
    <xsd:extension base="PathNode"/>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ElementLink -->
<!-- ~~~~~ -->
<xsd:complexType name="ElementLink">
  <xsd:complexContent>
    <xsd:extension base="GRLmodelElement">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="refs" type="xsd:IDREF"/> <!-- LinkRef -->
        <xsd:element name="dest" type="xsd:IDREF"/> <!-- GRLLinkableElement -->
        <xsd:element name="src" type="xsd:IDREF"/> <!-- GRLLinkableElement -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- EmptyPoint -->
<!-- ~~~~~ -->
<xsd:complexType name="EmptyPoint">
  <xsd:complexContent>
    <xsd:extension base="PathNode"/>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- EndPoint -->
<!-- ~~~~~ -->
<xsd:complexType name="EndPoint">
  <xsd:complexContent>
    <xsd:extension base="PathNode">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="outBindings" type="xsd:IDREF"/>
          <!-- OutBinding -->
        <xsd:element minOccurs="0" name="postcondition" type="Condition"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- EnumerationType -->
<!-- ~~~~~ -->
<xsd:complexType name="EnumerationType">
  <xsd:complexContent>
    <xsd:extension base="UCMmodelElement">
      <xsd:sequence>
        <xsd:element name="values" type="xsd:string"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="instances" type="xsd:IDREF"/> <!-- Variable -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Evaluation -->
<!-- ~~~~~ -->
<xsd:complexType name="Evaluation">
  <xsd:sequence>
    <xsd:element name="evaluation" type="xsd:integer"/>
    <xsd:element default="None" name="qualitativeEvaluation" type="QualitativeLabel"/>
    <xsd:element name="exceeds" type="xsd:boolean"/>
    <xsd:element minOccurs="0" name="indicatorEval" type="IndicatorEvaluation"/>
    <xsd:element name="intElement" type="xsd:IDREF"/> <!-- GRLContainableElement -->
    <xsd:element minOccurs="0" name="conversion" type="IndicatorConversion"/>
  </xsd:sequence>

```

```

</xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- EvaluationStrategy -->
<!-- ~~~~~ -->
<xsd:complexType name="EvaluationStrategy">
  <xsd:complexContent>
    <xsd:extension base="GRLmodelElement">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="evaluations" type="Evaluation"/>
        <xsd:element maxOccurs="unbounded" name="group" type="xsd:IDREF"/> <!-- StrategiesGroup -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="parentStrategies" type="xsd:IDREF"/>
          <!-- EvaluationStrategy -->
        <xsd:element minOccurs="0" name="includedStrategies" type="xsd:IDREFS"/>
          <!-- EvaluationStrategy { ordered} -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ExternalOperation -->
<!-- ~~~~~ -->
<xsd:complexType name="ExternalOperation">
  <xsd:complexContent>
    <xsd:extension base="ActiveResource">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="demands" type="xsd:IDREF"/> <!-- Demand -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- FailurePoint -->
<!-- ~~~~~ -->
<xsd:complexType name="FailurePoint">
  <xsd:complexContent>
    <xsd:extension base="PathNode">
      <xsd:sequence>
        <xsd:element name="failure" type="xsd:string"/>
        <xsd:element minOccurs="0" name="failureLabel" type="Label"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- GeneralResource -->
<!-- ~~~~~ -->
<xsd:complexType name="GeneralResource">
  <xsd:complexContent>
    <xsd:extension base="UCMmodelElement">
      <xsd:sequence>
        <xsd:element default="1" name="multiplicity" type="xsd:nonNegativeInteger"/>
        <xsd:element name="schedPolicy" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- GRLGraph -->
<!-- ~~~~~ -->
<xsd:complexType name="GRLGraph">
  <xsd:complexContent>
    <xsd:extension base="GRLmodelElement">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="connections" type="LinkRef"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="nodes" type="GRLNode"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="contRefs" type="ActorRef"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="comments" type="Comment"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>

```

```

</xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- GRLContainableElement -->
<!-- ~~~~~ -->
<xsd:complexType name="GRLContainableElement">
  <xsd:complexContent>
    <xsd:extension base="GRLLinkableElement">
      <xsd:sequence>
        <xsd:element minOccurs="0" name="actor" type="xsd:IDREF"/> <!-- Actor -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="refs" type="xsd:IDREF"/>
          <!-- IntentionalElementRef -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- GRLLinkableElement -->
<!-- ~~~~~ -->
<xsd:complexType name="GRLLinkableElement">
  <xsd:complexContent>
    <xsd:extension base="GRLmodelElement">
      <xsd:sequence>
        <xsd:element default="None" name="importance" type="ImportanceType"/>
        <xsd:element name="importanceQuantitative" type="xsd:integer"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="linksDest" type="xsd:IDREF"/> <!-- ElementLink -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="linksSrc" type="xsd:IDREF"/> <!-- ElementLink -->
        <xsd:element minOccurs="0" name="style" type="ConcreteStyle"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- GRLmodelElement -->
<!-- ~~~~~ -->
<xsd:complexType name="GRLmodelElement">
  <xsd:complexContent>
    <xsd:extension base="URNmodelElement"/>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- GRLNode -->
<!-- ~~~~~ -->
<xsd:complexType name="GRLNode">
  <xsd:complexContent>
    <xsd:extension base="GRLmodelElement">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="pred" type="xsd:IDREF"/> <!-- LinkRef -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="succ" type="xsd:IDREF"/> <!-- LinkRef -->
        <xsd:element minOccurs="0" name="contRef" type="xsd:IDREF"/> <!-- ActorRef -->
        <xsd:element name="pos" type="Position"/>
        <xsd:element name="size" type="Size"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- GRLspec -->
<!-- ~~~~~ -->
<xsd:complexType name="GRLspec">
  <xsd:sequence>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="intElements" type="GRLContainableElement"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="indConversions" type="IndicatorConversion"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="contribContextGroups" type="ContributionContextGroup"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="contribContexts" type="ContributionContext"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="actors" type="Actor"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="links" type="ElementLink"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="groups" type="StrategiesGroup"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="strategies" type="EvaluationStrategy"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="grlGraphs" type="GRLGraph"/>
  </xsd:sequence>

```

```

    <xsd:element minOccurs="0" name="info" type="ConcreteGRLspec"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- InBinding -->
<!-- ~~~~~ -->
<xsd:complexType name="InBinding">
  <xsd:sequence>
    <xsd:element name="id" type="xsd:ID" /> <!-- ADDED because InBinding is not a URNmodelElement (no ID) -->
    <xsd:element name="startPoint" type="xsd:IDREF"/> <!-- StartPoint -->
    <xsd:element name="stubEntry" type="xsd:IDREF"/> <!-- NodeConnection -->
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Indicator -->
<!-- ~~~~~ -->
<xsd:complexType name="Indicator">
  <xsd:complexContent>
    <xsd:extension base="GRLContainableElement">
      <xsd:sequence>
        <xsd:element name="unit" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- IndicatorConversion -->
<!-- ~~~~~ -->
<xsd:complexType name="IndicatorConversion">
  <xsd:complexContent>
    <xsd:extension base="GRLmodelElement">
      <xsd:sequence>
        <xsd:element name="unit" type="xsd:string"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="evals" type="xsd:IDREF"/>
        <!-- Evaluation -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- IndicatorEvaluation -->
<!-- ~~~~~ -->
<xsd:complexType name="IndicatorEvaluation">
  <xsd:sequence>
    <xsd:element name="realWorldValue" type="xsd:integer"/>
    <xsd:element name="realWorldLabel" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Initialization -->
<!-- ~~~~~ -->
<xsd:complexType name="Initialization">
  <xsd:sequence>
    <xsd:element name="value" type="xsd:string"/>
    <xsd:element name="variable" type="xsd:IDREF"/> <!-- Variable -->
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- IntentionalElement -->
<!-- ~~~~~ -->
<xsd:complexType name="IntentionalElement">
  <xsd:complexContent>
    <xsd:extension base="GRLContainableElement">
      <xsd:sequence>
        <xsd:element name="type" type="IntentionalElementType"/>
        <xsd:element default="AND" name="decompositionType" type="DecompositionType"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

```

```

<!-- ~~~~~ -->
<!-- IntentionalElementRef -->
<!-- ~~~~~ -->
<xsd:complexType name="IntentionalElementRef">
  <xsd:complexContent>
    <xsd:extension base="GRLNode">
      <xsd:sequence>
        <xsd:element name="def" type="xsd:IDREF"/> <!-- GRLContainableElement -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Label -->
<!-- ~~~~~ -->
<xsd:complexType name="Label">
  <xsd:sequence>
    <xsd:element name="deltaX" type="xsd:integer"/>
    <xsd:element name="deltaY" type="xsd:integer"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- LinearConversion -->
<!-- ~~~~~ -->
<xsd:complexType name="LinearConversion">
  <xsd:complexContent>
    <xsd:extension base="IndicatorConversion">
      <xsd:sequence>
        <xsd:element name="targetValue" type="xsd:integer"/>
        <xsd:element name="thresholdValue" type="xsd:integer"/>
        <xsd:element name="worstValue" type="xsd:integer"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- LinkRef -->
<!-- ~~~~~ -->
<xsd:complexType name="LinkRef">
  <xsd:complexContent>
    <xsd:extension base="GRLmodelElement">
      <xsd:sequence>
        <xsd:element name="curve" type="xsd:boolean"/>
        <xsd:element name="link" type="xsd:IDREF"/> <!-- ElementLink -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="bendpoints" type="LinkRefBendpoint"/>
          <!-- { ordered} -->
        <xsd:element minOccurs="0" name="label" type="Label"/>
        <xsd:element name="target" type="xsd:IDREF"/> <!-- GRLNode -->
        <xsd:element name="source" type="xsd:IDREF"/> <!-- GRLNode -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- LinkRefBendpoint -->
<!-- ~~~~~ -->
<xsd:complexType name="LinkRefBendpoint">
  <xsd:sequence>
    <xsd:element name="x" type="xsd:integer"/>
    <xsd:element name="y" type="xsd:integer"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Metadata -->
<!-- ~~~~~ -->
<xsd:complexType name="Metadata">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="value" type="xsd:string"/>
  </xsd:sequence>

```

```

</xsd:complexType>
<!-- ~~~~~ -->
<!-- NodeConnection -->
<!-- ~~~~~ -->
<xsd:complexType name="NodeConnection">
  <xsd:sequence>
    <xsd:element name="id" type="xsd:ID"/> <!-- ADDED because NodeConnection is not a URNmodelElement (no ID) -->
    <xsd:element default="100" name="probability" type="xsd:nonNegativeInteger"/>
    <xsd:element name="threshold" type="xsd:string"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="inBindings" type="xsd:IDREF"/> <!-- InBinding -->
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="outBindings" type="xsd:IDREF"/> <!-- OutBinding -->
    <xsd:element minOccurs="0" name="condition" type="Condition"/>
    <xsd:element minOccurs="0" name="timer" type="xsd:IDREF"/> <!-- Timer -->
    <xsd:element minOccurs="0" name="label" type="Label"/>
    <xsd:element name="target" type="xsd:IDREF"/> <!-- PathNode -->
    <xsd:element name="source" type="xsd:IDREF"/> <!-- PathNode -->
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- OWPeriodic -->
<!-- ~~~~~ -->
<xsd:complexType name="OWPeriodic">
  <xsd:complexContent>
    <xsd:extension base="OpenWorkload">
      <xsd:sequence>
        <xsd:element name="period" type="xsd:string"/>
        <xsd:element name="deviation" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- OWPhaseType -->
<!-- ~~~~~ -->
<xsd:complexType name="OWPhaseType">
  <xsd:complexContent>
    <xsd:extension base="OpenWorkload">
      <xsd:sequence>
        <xsd:element name="alpha" type="xsd:string"/>
        <xsd:element name="s" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- OWPoisson -->
<!-- ~~~~~ -->
<xsd:complexType name="OWPoisson">
  <xsd:complexContent>
    <xsd:extension base="OpenWorkload">
      <xsd:sequence>
        <xsd:element name="mean" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- OWUniform -->
<!-- ~~~~~ -->
<xsd:complexType name="OWUniform">
  <xsd:complexContent>
    <xsd:extension base="OpenWorkload">
      <xsd:sequence>
        <xsd:element name="start" type="xsd:string"/>
        <xsd:element name="end" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->

```

```

<!-- OpenWorkload -->
<!-- ~~~~~ -->
<xsd:complexType name="OpenWorkload">
  <xsd:complexContent>
    <xsd:extension base="Workload"/>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- OrFork -->
<!-- ~~~~~ -->
<xsd:complexType name="OrFork">
  <xsd:complexContent>
    <xsd:extension base="PathNode"/>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- OrJoin -->
<!-- ~~~~~ -->
<xsd:complexType name="OrJoin">
  <xsd:complexContent>
    <xsd:extension base="PathNode"/>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- OutBinding -->
<!-- ~~~~~ -->
<xsd:complexType name="OutBinding">
  <xsd:sequence>
    <xsd:element name="id" type="xsd:ID" /> <!-- ADDED because OutBinding is not a URNmodelElement (no ID) -->
    <xsd:element name="endPoint" type="xsd:IDREF"/> <!-- EndPoint -->
    <xsd:element name="stubExit" type="xsd:IDREF"/> <!-- NodeConnection -->
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- PassiveResource -->
<!-- ~~~~~ -->
<xsd:complexType name="PassiveResource">
  <xsd:complexContent>
    <xsd:extension base="GeneralResource">
      <xsd:sequence>
        <xsd:element minOccurs="0" name="component" type="xsd:IDREF"/> <!-- Component -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- PathNode -->
<!-- ~~~~~ -->
<xsd:complexType name="PathNode">
  <xsd:complexContent>
    <xsd:extension base="UCMmodelElement">
      <xsd:sequence>
        <xsd:element minOccurs="0" name="label" type="Label"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="pred" type="xsd:IDREF"/>
          <!-- NodeConnection -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="succ" type="xsd:IDREF"/>
          <!-- NodeConnection -->
        <xsd:element minOccurs="0" name="contRef" type="xsd:IDREF"/> <!-- ComponentRef -->
        <xsd:element minOccurs="0" name="pos" type="Position"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- PluginBinding -->
<!-- ~~~~~ -->
<xsd:complexType name="PluginBinding">
  <xsd:sequence>
    <xsd:element name="id" type="xsd:ID"/> <!-- ADDED because PluginBinding is not a URNmodelElement (no ID) -->
    <xsd:element default="100" name="probability" type="xsd:nonNegativeInteger"/>
  </xsd:sequence>

```

```

<xsd:element name="replicationFactor" type="xsd:string"/>
<xsd:element maxOccurs="unbounded" minOccurs="0" name="in" type="InBinding"/>
<xsd:element maxOccurs="unbounded" minOccurs="0" name="out" type="OutBinding"/>
<xsd:element name="plugin" type="xsd:IDREF"/> <!-- UCMmap -->
<xsd:element minOccurs="0" name="precondition" type="Condition"/>
<xsd:element maxOccurs="unbounded" minOccurs="0" name="components" type="ComponentBinding"/>
</xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Position -->
<!-- ~~~~~ -->
<xsd:complexType name="Position">
  <xsd:sequence>
    <xsd:element name="x" type="xsd:integer"/>
    <xsd:element name="y" type="xsd:integer"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ProcessingResource -->
<!-- ~~~~~ -->
<xsd:complexType name="ProcessingResource">
  <xsd:complexContent>
    <xsd:extension base="ActiveResource">
      <xsd:sequence>
        <xsd:element default="Processor" name="kind" type="DeviceKind"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="components" type="xsd:IDREF"/>
        <!-- Component -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- QualToQMapping -->
<!-- ~~~~~ -->
<xsd:complexType name="QualToQMapping">
  <xsd:sequence>
    <xsd:element name="realWorldLabel" type="xsd:string"/>
    <xsd:element name="evaluation" type="xsd:integer"/>
    <xsd:element name="qualitativeEvaluation" type="QualitativeLabel"/>
    <xsd:element name="exceeds" type="xsd:boolean"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- QualToQMappings -->
<!-- ~~~~~ -->
<xsd:complexType name="QualToQMappings">
  <xsd:complexContent>
    <xsd:extension base="IndicatorConversion">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="mappings" type="QualToQMapping"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Responsibility -->
<!-- ~~~~~ -->
<xsd:complexType name="Responsibility">
  <xsd:complexContent>
    <xsd:extension base="UCMmodelElement">
      <xsd:sequence>
        <xsd:element name="expression" type="xsd:string"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="demands" type="Demand"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="respRefs" type="xsd:IDREF"/> <!-- RespRef -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- RespRef -->

```

```

<!-- ~~~~~ -->
<xsd:complexType name="RespRef">
  <xsd:complexContent>
    <xsd:extension base="PathNode">
      <xsd:sequence>
        <xsd:element name="repetitionCount" type="xsd:string"/>
        <xsd:element name="hostDemand" type="xsd:string"/>
        <xsd:element name="respDef" type="xsd:IDREF"/> <!-- Responsibility -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ScenarioDef -->
<!-- ~~~~~ -->
<xsd:complexType name="ScenarioDef">
  <xsd:complexContent>
    <xsd:extension base="UCMmodelElement">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="initializations" type="Initialization"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="postconditions" type="Condition"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="preconditions" type="Condition"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="parentScenarios" type="xsd:IDREF"/>
        <!-- ScenarioDef -->
        <xsd:element minOccurs="0" name="includedScenarios" type="xsd:IDREFS"/> <!-- ScenarioDef { ordered} -->
        <xsd:element maxOccurs="unbounded" name="groups" type="xsd:IDREF"/> <!-- ScenarioGroup -->
        <xsd:element minOccurs="0" name="startPoints" type="xsd:IDREFS"/> <!-- StartPoint { ordered} -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="endPoints" type="xsd:IDREF"/>
        <!-- EndPoint -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ScenarioGroup -->
<!-- ~~~~~ -->
<xsd:complexType name="ScenarioGroup">
  <xsd:complexContent>
    <xsd:extension base="UCMmodelElement">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="scenarios" type="xsd:IDREF"/>
        <!-- ScenarioDef -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Size -->
<!-- ~~~~~ -->
<xsd:complexType name="Size">
  <xsd:sequence>
    <xsd:element name="width" type="xsd:integer"/>
    <xsd:element name="height" type="xsd:integer"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- StartPoint -->
<!-- ~~~~~ -->
<xsd:complexType name="StartPoint">
  <xsd:complexContent>
    <xsd:extension base="PathNode">
      <xsd:sequence>
        <xsd:element default="None" name="failureKind" type="FailureKind"/>
        <xsd:element name="failureList" type="xsd:string"/>
        <xsd:element minOccurs="0" name="failureLabel" type="Label"/>
        <xsd:element minOccurs="0" name="workload" type="Workload"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="inBindings" type="xsd:IDREF"/>
        <!-- InBinding -->
        <xsd:element minOccurs="0" name="precondition" type="Condition"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

```

```

</xsd:extension>
</xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- StrategiesGroup -->
<!-- ~~~~~ -->
<xsd:complexType name="StrategiesGroup">
  <xsd:complexContent>
    <xsd:extension base="GRLmodelElement">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="strategies" type="xsd:IDREF"/>
          <!-- EvaluationStrategy -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Stub -->
<!-- ~~~~~ -->
<xsd:complexType name="Stub">
  <xsd:complexContent>
    <xsd:extension base="PathNode">
      <xsd:sequence>
        <xsd:element name="dynamic" type="xsd:boolean"/>
        <xsd:element name="synchronizing" type="xsd:boolean"/>
        <xsd:element name="blocking" type="xsd:boolean"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="bindings" type="PluginBinding"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Timer -->
<!-- ~~~~~ -->
<xsd:complexType name="Timer">
  <xsd:complexContent>
    <xsd:extension base="WaitingPlace">
      <xsd:sequence>
        <xsd:element minOccurs="0" name="timeoutPath" type="xsd:IDREF"/> <!-- NodeConnection -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- UCMmap -->
<!-- ~~~~~ -->
<xsd:complexType name="UCMmap">
  <xsd:complexContent>
    <xsd:extension base="UCMmodelElement">
      <xsd:sequence>
        <xsd:element name="singleton" type="xsd:boolean"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="parentStub" type="xsd:IDREF"/>
          <!-- PluginBinding -->
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="contRefs" type="ComponentRef"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="connections" type="NodeConnection"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="nodes" type="PathNode"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="comments" type="Comment"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- UCMmodelElement -->
<!-- ~~~~~ -->
<xsd:complexType name="UCMmodelElement">
  <xsd:complexContent>
    <xsd:extension base="URNmodelElement"/>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->

```

```

<!-- UCMspec -->
<!-- ~~~~~ -->
<xsd:complexType name="UCMspec">
  <xsd:sequence>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="enumerationTypes" type="EnumerationType"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="variables" type="Variable"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="scenarioGroups" type="ScenarioGroup"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="resources" type="GeneralResource"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="ucmMaps" type="UCMmap"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="components" type="Component"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="componentTypes" type="ComponentType"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="responsibilities" type="Responsibility"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="scenarioDefs" type="ScenarioDef"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- URNlink -->
<!-- ~~~~~ -->
<xsd:complexType name="URNlink">
  <xsd:complexContent>
    <xsd:extension base="URNmodelElement">
      <xsd:sequence>
        <xsd:element name="type" type="xsd:string"/>
        <xsd:element name="toElem" type="xsd:IDREF"/> <!-- URNmodelElement -->
        <xsd:element name="fromElem" type="xsd:IDREF"/> <!-- URNmodelElement -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- URNmodelElement -->
<!-- ~~~~~ -->
<xsd:complexType name="URNmodelElement">
  <xsd:sequence>
    <xsd:element name="id" type="xsd:ID"/>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="metadata" type="Metadata"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="toLinks" type="xsd:IDREF"/> <!-- URNlink -->
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="fromLinks" type="xsd:IDREF"/> <!-- URNlink -->
    <xsd:element minOccurs="0" name="desc" type="Description"/>
    <xsd:element minOccurs="0" name="concern" type="xsd:IDREF"/> <!-- Concern -->
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- URNspec -->
<!-- ~~~~~ -->
<xsd:complexType name="URNspec">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element minOccurs="0" name="ucmspec" type="UCMspec"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="metadata" type="Metadata"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="urnLinks" type="URNlink"/>
    <xsd:element minOccurs="0" name="grlSpec" type="GRLspec"/>
    <xsd:element minOccurs="0" name="info" type="ConcreteURNspec"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="concerns" type="Concern"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Variable -->
<!-- ~~~~~ -->
<xsd:complexType name="Variable">
  <xsd:complexContent>
    <xsd:extension base="UCMmodelElement">
      <xsd:sequence>
        <xsd:element default="Boolean" name="type" type="DatatypeKind"/>
        <xsd:element minOccurs="0" name="enumerationType" type="xsd:IDREF"/> <!-- EnumerationType -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

```

```

<!-- ~~~~~ -->
<!-- WaitingPlace -->
<!-- ~~~~~ -->
<xsd:complexType name="WaitingPlace">
  <xsd:complexContent>
    <xsd:extension base="PathNode">
      <xsd:sequence>
        <xsd:element name="waitType" type="WaitKind"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- Workload -->
<!-- ~~~~~ -->
<xsd:complexType name="Workload">
  <xsd:complexContent>
    <xsd:extension base="UCMmodelElement">
      <xsd:sequence>
        <xsd:element default="ms" name="unit" type="TimeUnit"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
</xsd:schema>

```

## Annex B

### Textual URN specification

(This annex forms an integral part of this Recommendation.)

#### B.1 Introduction to the Textual URN specification

This annex describes the textual syntax for the User Requirements Notation, called TURN (Textual URN). For the purpose of this annex, the URN definitions in clauses 6 to 8 are referred to as URN, GRL, or UCM, while the textual syntax defined in this annex is referred to as the Textual URN, Textual GRL, or Textual UCM. The annex shows an example for GRL and UCM, respectively, and their corresponding TURN specification. In addition, the TURN grammar is specified.

The main objective of TURN is to support the modeling of very large URN specifications, where scalability issues have been encountered with the graphical syntax. Navigating thousands of separate maps or goal graphs as distinct diagrams becomes unwieldy. Similarly, a map with many paths that cannot be shown on a single screen or printed page becomes difficult to understand. In addition, the entering of large specifications in graphical tools has proven tedious, as the user must be concerned with layout issues that are unrelated to the information that is attempted to be modeled. In general, TURN offers an alternative input medium for URN specifications. This is also useful for small URN specification, where the modeler may choose to use both representations, i.e. the textual syntax which is typically faster for entering a specification and the graphical syntax which is typically faster for understanding small to medium-sized specifications.

The TURN grammar strikes a balance between supporting as many language features of URN as possible and the usability, convenience, and expediency of the textual syntax. Not all URN concepts are supported. The differences are highlighted and a mapping is described from the TURN grammar to the abstract grammar of URN. The textual syntax for GRL covers the abstract grammar of GRL in its entirety. Collapsed actor references from the concrete grammar of GRL are not covered. The textual syntax for UCM does not cover performance concepts, component types, descriptions of model elements (all not an essential part of the UCM notation), and empty points from the abstract grammar of UCM but everything else from the abstract grammar of UCM. The first two are not essential to a URN specification, while empty points are covered indirectly and comments may be used for descriptions. Direction arrows from the concrete grammar of UCM are not supported.

Logically a TURN specification is a single body of text that for handling purposes may be split into more than one file, but how multiple files are treated as a single body is a methodology and/or tool issue beyond the scope of this Recommendation. Furthermore, even though this annex only uses keywords with lower case characters in the TURN grammar, all keywords are case-insensitive.

All constraints and default values specified in clauses 6, 7, and 8 also apply to a TURN specification unless stated otherwise and only if the corresponding metaclass is covered by TURN. In the TURN grammar, referenced elements are identified by a <Name> instead of the id attribute used in URN as explained in more detail in B.2. The <Name> is used instead of the numerical id attribute, because the unique id attribute is automatically assigned to ensure uniqueness and not directly used by a modeller. Having to explicitly keep track of such numbers is error-prone and inconvenient.

The TURN grammar in this annex builds on the grammar defined in clause 9. Text conforming to the grammar defined in clause 9 is used as the Text describing the detail of conditions, responsibility actions, or metadata.

As with all grammars defined in this Recommendation, the conventions of [ITU-T Z.111] also apply to this annex. The mapping from TURN to URN adheres to the following general guidelines. TURN production rules are mapped by default to the URN metaclass with the same name as the TURN

production rule including any attributes with the same name. The mapping of the Textual URN to the abstract syntax is given in English text augmenting the grammar.

Comments in TURN are in C++/Java style, supporting both bounded comments (enclosed by `"/**` and `"/**`, which can be nested) and terminated comments which extend from `"/**` to the end of the current line.

## B.2 User Requirements Notation: core concepts

The TURN grammar in this section covers clause 6, i.e. the core concepts of URN. All concepts are covered by TURN. Listing B.1 shows the grammar interspersed with explanatory text that introduces the Textual URN by example, while Listing B.2 gives a concrete example TURN specification of the core concepts of URN. For the abstract grammar of a URN model to be complete, additional links have to be created between the objects of the constructed URN model. These links can be inferred from the abstract grammar and are not further described.

### Listing B.1(a) – Grammar for core concepts of Textual User Requirements Notation

```
<URNspec> ::= urnModel <TURNidentifier>
  [<ConcreteURNspec>]
  {<GRLspec> | <UCMspec> | <Concern> | <URNlink> | <Metadata>}*

<ConcreteURNspec> ::= <left curly bracket>
  [description <String>]
  [author <String>]
  [created <String>]
  [modified <String>]
  [version <String>]
  [urnVersion <String>]
<right curly bracket>
```

A `<URNspec>` represents a URNspec. `<TURNidentifier>` represents URNspec.name.

A `<ConcreteURNspec>` represents a ConcreteURNSpec. 'description', 'author', 'created', 'modified', 'version', and 'urnVersion' map to the attributes of ConcreteURNSpec with the same name.

For more information on `<GRLspec>`, see clause B.3 on the Goal-oriented Requirement Language. For more information on `<UCMspec>`, see Section B.4 on Use Case Maps.

### Listing B.1(b) – Grammar for core concepts of TURN (continued)

```
<Concern> ::= concern <Name> <colon>
  [<Condition>]
  <QualifiedReferenceToURNmodelElement> {<comma>
  <QualifiedReferenceToURNmodelElement>}*
```

A `<Concern>` represents a Concern. `<Name>` is mapped to Concern.name according to the rules specified for `<Name>`. `<QualifiedReferenceToURNmodelElement>` items uniquely identify Concern.elements. The optional `<Condition>` represents Concern.condition.

### Listing B.1(c) – Grammar for core concepts of TURN (continued)

```
<Condition> ::=
  {<left square bracket> <BooleanExpression> <right square bracket>}
```

A `<Condition>` represents a Condition. `<BooleanExpression>` represents Condition.expression.

### Listing B.1(d) – Grammar for core concepts of TURN (continued)

```
<URNlink> ::= link [<Name> <colon>]
             [<left square bracket> <Text> <right square bracket>]
             <QualifiedReferenceToURNmodelElement> <long arrow>
             <QualifiedReferenceToURNmodelElement>
```

A <URNlink> represents a URNlink. <Name> is mapped to URNlink.name according to the rules specified for <Name>. If <Name> is not specified, the name is empty. <Text> represents URNlink.type. The first <QualifiedReferenceToURNmodelElement> uniquely identifies URNlink.fromElem. The second <QualifiedReferenceToURNmodelElement> uniquely identifies URNlink.toElem.

### Listing B.1(e) – Grammar for core concepts of TURN (continued)

```
<Metadata> ::=
             {metadata | {<QualifiedReferenceToURNmodelElement> <colon>}}
             <identifier> <equal sign> <left square bracket> <Text> <right square
             bracket>
```

A <Metadata> represents a Metadata. When a <QualifiedReferenceToURNmodelElement> is specified, then the <Metadata> is contained in the referenced URNmodelElement. When 'metadata' is present, the <Metadata> is contained in the URNspec (i.e. it is for the URNspec). <identifier> represents Metadata.name. <Text> represents Metadata.value.

### Listing B.1(f) – Grammar for core concepts of TURN (continued)

```
<QualifiedReferenceToURNmodelElement> ::= <QualifiedTURNidentifier>
<QualifiedTURNidentifier> ::= {<Name> <full stop>}* <Name>
```

The <QualifiedTURNidentifier> of a <QualifiedReferenceToURNmodelElement> uniquely identifies the referenced URNmodelElement (i.e. a <Concern>, <URNlink>, <Actor>, <GRLContainableElement> (<IntentionalElement>, <Indicator>), <ElementLink> (<Contribution>, <Decomposition>, <Dependency>), <StrategiesGroup>, <EvaluationStrategy>, <IndicatorConversion> (<LinearConversion>, <QualToQMappings>), <ContributionContextGroup>, <ContributionContext>, <UCMmap>, <StartPoint>, <RespRef>, <OrJoin>, <AndJoin>, <WaitingPlace>, <Timer>, <FailurePoint>, <OrFork>, <AndFork>, <Stub>, <EndPoint>, <ComponentRef>, <ScenarioGroup>, <ScenarioDef>, <Variable>, or <EnumerationType>).

Since the names of some URN model elements may be the same (e.g., start points and end points on different maps), a <QualifiedTURNidentifier> includes the name of the container of a model element. The left <Name> is always the container of the right <Name> for any pair of adjacent <Name> items separated by a <full stop> (i.e., several nested containers may be specified).

### Listing B.1(g) – Grammar for core concepts of TURN (continued)

```
<Name> ::= {<TURNidentifier> [<hash sign> <DisplayName>]} | [<DisplayName>]
<TURNidentifier> ::= <identifier> | <String>
<DisplayName> ::= <identifier> | <String>
```

A <Name> identifies a model element. The <TURNidentifier> represents the unique identifier of the model element in the TURN specification. If <TURNidentifier> is not specified, then the <DisplayName> is used as the unique identifier. The <DisplayName> represents the name of the model element. If the <DisplayName> is not specified, then the <TURNidentifier> is also used for the name of the model element. The motivation behind <Name> is to allow elements to be identified

and referenced easily within a TURN specification using the shorter <TURNIdentifier> instead of the longer <DisplayName>. Furthermore, <RespRef> and <ComponentRef> make use of the <Name> to also identify the associated Responsibility and Component, respectively, as explained below.

<identifier> is defined in clause 9. Any keyword of the Textual URN may not be used as an <identifier> in TURN, as a <String> in <TURNIdentifier>, and as a <String> in <DisplayName>.

The following examples illustrate the three cases for <Name>.

- 1) Full name (e.g., fwd1#"forward Signal" or fwd1#forwardSignal or "forward 1"#"forward Signal" or "forward 1"#"forwardSignal)
2. <DisplayName> only (e.g., "forward Signal" or forwardSignal)
3. <TURNIdentifier> only (e.g., fwd1 or "forward 1")

Case 1 introduces a short <TURNIdentifier> to be used in references instead of having to write the long <DisplayName> all the time. The name of the model element is the <DisplayName>. The unique identifier of the model element in the TURN specification is the <TURNIdentifier>.

For <RespRef> and <ComponentRef>, the <TURNIdentifier> is the unique identifier of the <RespRef> or <ComponentRef>, the <DisplayName> is the unique identifier of the associated Responsibility (i.e., RespRef.respDef) or Component (i.e., ComponentRef.compDef), and the <DisplayName> is the name of the <RespRef> or <ComponentRef> as well as the name of their associated Responsibility or Component, respectively.

For Case 2, the model element is identified and referenced with its <DisplayName>, i.e. the long name is used for referencing. Hence, the <DisplayName> is used as the unique identifier of the model element in the TURN specification. The name of the model element is also the <DisplayName>.

For <RespRef> and <ComponentRef>, the <RespRef> and <ComponentRef> does not need to be uniquely identified and hence the <DisplayName> is the unique identifier of the associated Responsibility (i.e., RespRef.respDef) or Component (i.e., ComponentRef.compDef). The <DisplayName> is also the name of the <RespRef> or <ComponentRef> as well as the name of their associated Responsibility or Component, respectively.

For Case 3, the model element is identified and referenced with its <TURNIdentifier>, i.e., a long name is not required for this particular model element. Hence, the <TURNIdentifier> is used as the unique identifier of the model element in the TURN specification. The name of the model element is also the <TURNIdentifier>.

For <RespRef> and <ComponentRef>, the <RespRef> and <ComponentRef> does not need to be uniquely identified and hence the <TURNIdentifier> is the unique identifier of the associated Responsibility (i.e., RespRef.respDef) or Component (i.e., ComponentRef.compDef). The <TURNIdentifier> is also the name of the <RespRef> or <ComponentRef> as well as the name of their associated Responsibility or Component, respectively.

A <Name> does not have to be globally unique. The following rules apply: (i) the <Name> of each top-level element in the core (<Concern>, <URNLink>), in the GRL specification (<Actor>, <StrategiesGroup>, <EvaluationStrategy>, <IndicatorConversion>, <ContributionContextGroup>, <ContributionContext>; see clause B.3), and the UCM specification (<UCMmap>, <ScenarioGroup>, <ScenarioDef>, <Variable>, <EnumerationType>; see clause B.4) must be unique; (ii) the <Name> of each referencable element within a GRL <Actor> must be unique; (iii) the <Name> of each referencable element within a <UCMmap> must be unique with the exception of <RespRef> items and <EndPoint> items. A <Name> is unique because either the <TURNIdentifier> is unique or the <DisplayName> is unique if <TURNIdentifier> is not used.

If a <Name> occurs in a definition (i.e., not in a production rule that contains "ReferenceTo"), and no <hash sign> is present, the <Name> is interpreted as <TURNIdentifier>. The name of the model element is then considered to be identical to the unique identifier of the model element. If a <Name>

occurs in a reference (i.e., a production rule that contains "ReferenceTo"), the name is first interpreted as a <TURNidentifier> and the specification is searched for a matching model element. If a model element with a matching identifier is found, the <Name> refers to that model element. If no matching model element is found, the <Name> is interpreted as <DisplayName>, and the specification is searched for a matching model element. If a model element with a matching name is found, the <Name> refers to that model element. If no matching model element is found, the specification is not well-formed. A <hash sign> must not be present in a reference.

When an <EndPoint>, <StartPoint>, <WaitingPlace>, <Timer>, or <FailurePoint> does not have a name, an anonymous name is used as the identifier of the PathNode and the empty string is used as its name.

### Listing B.1(h) – Grammar for core concepts of TURN (continued)

```

<Text> ::= <not right square bracket>+
<String> ::= <quotation mark>
           <not quotation mark or space> <not quotation mark>*
           <quotation mark>
<BooleanExpression> ::= <expression>

<not right square bracket> ::= (any printable character or space character
                               recommended in UCS [ITU-T T.55] except RIGHT SQUARE BRACKET)
<not quotation mark or space> ::= (any printable character recommended in UCS
                                   [ITU-T T.55] except QUOTATION MARK)
<not quotation mark> ::= (any printable character or space character recommended in
                           UCS [ITU-T T.55] except QUOTATION MARK)

<left curly bracket> ::= {
<right curly bracket> ::= }
<colon> ::= :
<comma> ::= ,
<left square bracket> ::= [
<right square bracket> ::= ]
<long arrow> ::= -->
<equal sign> ::= =
<full stop> ::= .
<hash sign> ::= #
<quotation mark> ::= "

```

A <BooleanExpression> is an <expression> whose computed type is Boolean as defined in clause 9.3. <expression> and <integer literal> are defined in clause 9. Note that Listing B.1(h) shows all lexical definitions required for the textual syntax of the core concepts of URN, but also for the Textual GRL (see B.3) and Textual UCM (see B.4).

### Listing B.2 – Example TURN for URN core concepts

```

urnModel Example {
    description "This is an example."
    author "ITU-T"
    created "Feb-20-2017"
    modified "Feb-20-2017"
    version "1.0"
    urnVersion "29.0"
}

... // see section for Goal-oriented Requirement Language

... // see section for Use Case Maps

concern TCConcern : TelP

```

```

link [trace] TelP --> OAgent
link [trace] TelP --> TAgent

TCCConcern : CompanyValue = [High]

```

### B.3 Goal-oriented Requirement Language

The TURN grammar in this section covers clause 7, i.e. the GRL concepts of URN. All concepts in the abstract grammar of GRL are covered by TURN. Listing B.3 gives a concrete example TURN specification of the GRL concepts of URN, showing a possible textual syntax for the graphical URN model in Figure B.1. Listing B.4 shows the grammar interspersed with explanatory text that introduces the Textual GRL by example.

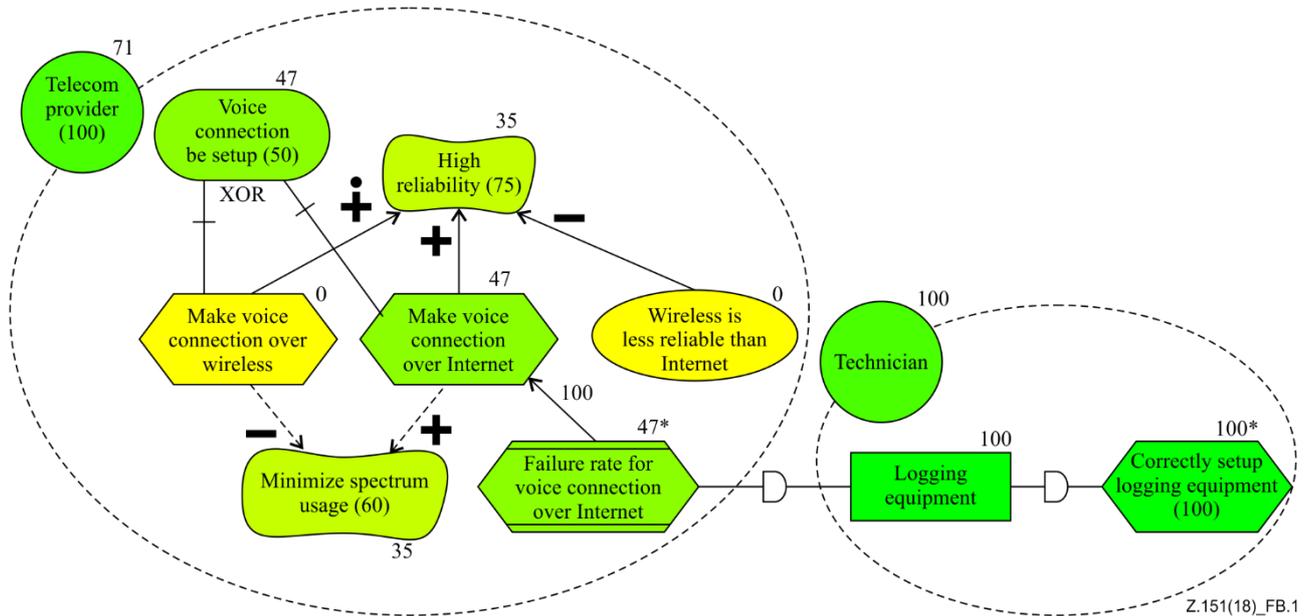


Figure B.1 – Example GRL model

While the TURN grammar matches the GRL metamodel to a large degree, there are some minor differences:

1. The TURN grammar does not allow links among actors to be specified. However, this is more of a graphical visualization issue than a specification issue and hence not as useful for a TURN specification.
2. The TURN grammar requires an intentional element to be specified inside an actor, which results in a well-nested specification that is easier to comprehend and maintain. Intentional elements outside an actor are not allowed.
3. The TURN grammar allows either qualitative or quantitative importance/contribution values to be specified for an element but not both at the same time. However, this restriction reflects the spirit of URN.

Listing B.3 – TURN for example GRL model from Figure B.1

```

actor TelP#"Telecom Provider" {
  importance 100
  goal VoiceConn#"Voice Connection Be Setup" {
    importance 50
  }
  softgoal HighRel#"High Reliability" {

```

```

        description "This is the most important objective of the stakeholder."
        importance 75
    }
    softgoal SpecUsage#"Minimize Spectrum Usage" {
        importance 60
    }
    task MakeVoiceOverInternet#"Make Voice Connection Over Internet" {
        contributesTo HighRel with somePositive
        contributesTo SpecUsage correlated with somePositive
        xor decomposes VoiceConn
    }
    task MakeVoiceOverWireless#"Make Voice Connection Over Wireless" {
        contWirelessVoiceConnToHighRel contributesTo HighRel with make
        contributesTo SpecUsage correlated with someNegative
        xor decomposes VoiceConn
    }
    indicator VoiceConnFailureRate#"Failure Rate for Voice ConnectionOver
    Internet" {
        unit "failures/week/10000 connections"
        contVoiceConnFailureRateToInternetVoiceConn contributesTo
            MakeVoiceOverInternet with 100
        dependsOn Tech.LoggEquip
    }
    belief WirelessReliability#"Wireless is less reliable than Internet" {
        contributesTo HighRel with SomeNegative
    }
}
actor Tech#"Technician" {
    resource LoggEquip#"Logging Equipment" {
        dependsOn EquipSetup
    }
    task EquipSetup#"Correctly setup logging equipment" {
        importance 100
    }
}
strategiesGroup SG1 : Eval1
strategy Eval1#"Internet Connection" {
    author "ITU-T"
    Tech.EquipSetup evaluation 100
    TelP.VoiceConnFailureRate real 265 convertedWith LC1
}
linearConversion LC1#"Weakly Failures" {
    unit "failures/week/10000 connections"
    target 0
    threshold 500
    worst 10000
}
// The following mappingConversion is an alternative to the above linearConversion
// as
// only one may be specified for the same element (i.e. VoiceConnFailureRate). If
// the
// mappingConversion is used, then the mappingConversion replaces the
// linearConversion in strategy Eval1 (TelP.VoiceConnFailureRate {real "Class 2"
// convertedWith MC1}). Note that Class 2 in Table 3 at the end of clause 7.6 is
// assumed to map to 47 instead of 25 to match the example in Figure B.1.
MappingConversion MC1#"EquipmentClassification" {
    unit "equipment class"
    real "Class 1" --> 100
    real "Class 2" --> 47
    real "Class 3" --> -25
}
// These examples of contributionContexts describe the situation where (a) the make

```

```

// contribution in Figure B.2 is changed to a help contribution (as described by
the
// Pessimistic context) and (b) the 100 contribution is additionally changed to 50
// (as described by the PessimisticIneffective context).
contributionContextGroup CCG1 : CCP, CCPI
contributionContext CCP#"Pessimistic" {
    contWirelessVoiceConnToHighRel with help
}
contributionContext CCPI#"PessimisticIneffective" {
    contVoiceConnFailureRateToInternetVoiceConn with 50
    includes Pessimistic
}
}

```

Listing B.4 provides the grammar for the Textual GRL. The text between the listings describes the mapping from the textual syntax to the abstract grammar of GRL. The mapping is provided by describing the element in the abstract grammar of GRL that are created corresponding to the concrete syntax elements encountered in a Textual GRL specification. For the abstract grammar of a GRL model to be complete, additional links have to be created between the objects of the constructed GRL model. These links can be inferred from the abstract grammar and are not further described.

#### Listing B.4(a) – Grammar for Textual GRL

```

<GRLspec> ::=
    [showAsMeansEnd]
    {<Actor> | <StrategiesGroup> | <EvaluationStrategy> | <IndicatorConversion>
    | <ContributionContextGroup> | <ContributionContext>}*

```

A <GRLspec> represents a GRLspec.

If 'showAsMeansEnd' is present, GRLspec.info.showAsMeansEnd is true; otherwise it is false.

#### Listing B.4(b) – Grammar for Textual GRL (continued)

```

<Actor> ::= actor <Name> <left curly bracket>
    [importance {<ImportanceType> | <QuantitativeValue>}]
    <GRLContainableElement>*
<right curly bracket>

<ImportanceType> ::= high | medium | low | none

```

An <Actor> represents an Actor. <Name> is mapped to Actor.name according to the rules specified for <Name>. <ImportanceType> represents Actor.importance. If <ImportanceType> is 'high', then Actor.importance is High. If <ImportanceType> is 'medium', then Actor.importance is Medium. If <ImportanceType> is 'low', then Actor.importance is Low. If <ImportanceType> is 'none', then Actor.importance is None. <QuantitativeValue> represents Actor.importanceQuantitative.

#### Listing B.4(c) – Grammar for Textual GRL (continued)

```

<GRLContainableElement> ::=
    <IntentionalElement> | <Indicator>

<IntentionalElement> ::= <IntentionalElementType> <Name> <left curly bracket>
    [importance {<ImportanceType> | <QuantitativeValue>}]
    <ElementLink>*
<right curly bracket>

<IntentionalElementType> ::= softgoal | goal | task | resource | belief

<Indicator> ::= indicator <Name> <left curly bracket>
    [importance {<ImportanceType> | <QuantitativeValue>}]

```

```

    [unit <String>]
    <ElementLink>*
<right curly bracket>

```

A <GRLContainableElement> represents a GRLContainableElement. An <IntentionalElement> represents an IntentionalElement. An <Indicator> represents an Indicator. Note that the <DecompositionType> is specified with <Decomposition>.

For <IntentionalElement>, <Name> is mapped to IntentionalElement.name according to the rules specified for <Name>. <IntentionalElementType> represents IntentionalElement.type. If <IntentionalElementType> is 'softgoal', 'goal', 'task', 'resource', or 'belief', then IntentionalElement.type is Softgoal, Goal, Task, Resource, or Belief, respectively. <ImportanceType> represents IntentionalElement.importance. If <ImportanceType> is 'high', then IntentionalElement.importance is High. If <ImportanceType> is 'medium', then IntentionalElement.importance is Medium. If <ImportanceType> is 'low', then IntentionalElement.importance is Low. If <ImportanceType> is 'none', then IntentionalElement.importance is None. <QuantitativeValue> represents IntentionalElement.importanceQuantitative.

For <Indicator>, <Name> is mapped to Indicator.name according to the rules specified for <Name>. <ImportanceType> represents Indicator.importance. If <ImportanceType> is 'high', then Indicator.importance is High. If <ImportanceType> is 'medium', then Indicator.importance is Medium. If <ImportanceType> is 'low', then Indicator.importance is Low. If <ImportanceType> is 'none', then Indicator.importance is None. <QuantitativeValue> represents Indicator.importanceQuantitative. The unit of an Indicator is covered by the 'unit' <String>.

#### Listing B.4(d) – Grammar for Textual GRL (continued)

```

<ElementLink> ::=
    <Contribution> | <Decomposition> | <Dependency>

<Contribution> ::= [<Name>]
    contributesTo <ReferenceToGRLContainableElement>
    [correlated] with {<ContributionType> | <QuantitativeValue>}

<ContributionType> ::= make | help | somePositive | unknown | someNegative | hurt |
    break

```

An <ElementLink> represents an ElementLink.

A <Contribution> represents a Contribution. <Name> is mapped to Contribution.name according to the rules specified for <Name>. If <Name> is not specified, the name is empty. <ReferenceToGRLContainableElement> uniquely identifies Contribution.dest. Contribution.src is the IntentionalElement or Indicator represented by the <GRLContainableElement> in which this <Contribution> is contained. <QuantitativeValue> represents Contribution.quantitativeContribution. If 'correlated' is present, Contribution.correlation is true; otherwise it is false. If <ContributionType> is 'make', 'help', 'somePositive', 'unknown', 'someNegative', 'hurt', or 'break', then Contribution.contribution is Make, Help, SomePositive, Unknown, SomeNegative, Hurt, or Break, respectively.

#### Listing B.4(e) – Grammar for Textual GRL (continued)

```

<Decomposition> ::= [<Name>]
    <DecompositionType> decomposes <ReferenceToGRLContainableElement>

<DecompositionType> ::= and | or | xor

```

A <Decomposition> represents a Decomposition. <Name> is mapped to Decomposition.name according to the rules specified for <Name>. If <Name> is not specified, the name is empty. <ReferenceToGRLContainableElement> uniquely identifies Decomposition.dest. Decomposition.src is the IntentionalElement or Indicator represented by the <GRLContainableElement> in which this <Decomposition> is contained. If <DecompositionType> is 'and', then Decomposition.dest.decompositionType is AND. If <DecompositionType> is 'or', then Decomposition.dest.decompositionType is OR. If <DecompositionType> is 'xor', then Decomposition.dest.decompositionType is XOR.

If two <Decomposition> items exist that references the same <GRLContainableElement> and the <DecompositionType> of the <Decomposition> items is not the same, then the specification is not well-formed.

#### Listing B.4(f) – Grammar for Textual GRL (continued)

```
<Dependency> ::= [<Name>]
                dependsOn <ReferenceToGRLContainableElement>
```

A <Dependency> represents a Dependency. <Name> is mapped to Dependency.name according to the rules specified for <Name>. If <Name> is not specified, the name is empty. <ReferenceToGRLContainableElement> uniquely identifies Dependency.dest. Dependency.src is the IntentionalElement or Indicator represented by the <GRLContainableElement> in which this <Dependency> is contained.

#### Listing B.4(g) – Grammar for Textual GRL (continued)

```
<StrategiesGroup> ::= strategiesGroup <Name> <colon>
                    <ReferenceToEvaluationStrategy> {<comma> <ReferenceToEvaluationStrategy>}*

<EvaluationStrategy> ::= strategy <Name> <left curly bracket>
                        [<ConcreteStrategy>]
                        <Evaluation>*
                        [includes <ReferenceToEvaluationStrategy> {<comma>
                            <ReferenceToEvaluationStrategy>}*]
                        <right curly bracket>

<ConcreteStrategy> ::=
                    author <String>
```

A <StrategiesGroup> represents a StrategiesGroup. <Name> is mapped to StrategiesGroup.name according to the rules specified for <Name>. <ReferenceToEvaluationStrategy> items uniquely identify StrategiesGroup.strategies.

An <EvaluationStrategy> represents a EvaluationStrategy. <Name> is mapped to EvaluationStrategy.name according to the rules specified for <Name>. Its <ConcreteStrategy> represents EvaluationStrategy.info. Its <Evaluation> items represent EvaluationStrategy.evaluations. <ReferenceToEvaluationStrategy> items uniquely identify EvaluationStrategy.includedStrategies.

A <ConcreteStrategy> represents a ConcreteStrategy. The 'author' <String> represent ConcreteStrategy.author.

#### Listing B.4(h) – Grammar for Textual GRL (continued)

```
<Evaluation> ::=
    {<QualifiedReferenceToIntentionalElement> [exceeding]
      evaluation <QualitativeLabel> | <QuantitativeValue>}}
    | {<QualifiedReferenceToIndicator> [exceeding]
      <IndicatorEvaluation> convertedWith <ReferenceToIndicatorConversion>}
```

```

<QualitativeLabel> ::= denied | weaklyDenied | weaklySatisfied | satisfied |
                    conflict | unknown | none

<IndicatorEvaluation> ::=
    real {<String> | <RealWorldValue>}

```

An <Evaluation> represents an Evaluation. <QualifiedReferenceToIntentionalElement> and <QualifiedReferenceToIndicator> uniquely identify Evaluation.intElement. If 'exceeding' is present, Evaluation.exceeds is true; otherwise it is false. <QualitativeLabel> represents Evaluation.qualitativeEvaluation. If <QualitativeLabel> is 'denied', 'weaklyDenied', 'weaklySatisfied', 'satisfied', 'conflict', 'unknown', or 'none', then Evaluation.qualitativeEvaluation is Denied, WeaklyDenied, WeaklySatisfied, Satisfied, Conflict, Unknown, or None, respectively. <QuantitativeValue> maps to Evaluation.evaluation. Its <IndicatorEvaluation> represents Evaluation.indicatorEval. <ReferenceToIndicatorConversion> uniquely identifies Evaluation.conversion.

An <IndicatorEvaluation> represents an IndicatorEvaluation. <String> represents IndicatorEvaluation.realWorldLabel. <RealWorldValue> represents IndicatorEvaluation.realWorldValue.

#### Listing B.4(i) – Grammar for Textual GRL (continued)

```

<IndicatorConversion> ::=
    <LinearConversion> | <QualToQMappings>

<LinearConversion> ::= linearConversion <Name> <left curly bracket>
    unit <String>
    target <RealWorldValue>
    threshold <RealWorldValue>
    worst <RealWorldValue>
<right curly bracket>

<QualToQMappings> ::= mappingConversion <Name> <left curly bracket>
    unit <String>
    <QualToQMMapping>*
<right curly bracket>

<QualToQMMapping> ::=
    [exceeding] real <String> <long arrow> {<QualitativeLabel> |
    <QuantitativeValue>}

```

An <IndicatorConversion> represents an IndicatorConversion.

A <LinearConversion> represents a LinearConversion. <Name> is mapped to LinearConversion.name according to the rules specified for <Name>. The 'unit' <String> represents LinearConversion.unit. The 'target' <RealWorldValue> represents LinearConversion.targetValue. The 'threshold' <RealWorldValue> represents LinearConversion.thresholdValue. The 'worst' <RealWorldValue> represents LinearConversion.worstValue.

A <QualToQMappings> represents a QualToQMappings. <Name> is mapped to QualToQMappings.name according to the rules specified for <Name>. The 'unit' <String> represents QualToQMappings.unit. Its <QualToQMMapping> items represent QualToQMappings.mappings.

A <QualToQMMapping> represents a QualToQMMapping. If 'exceeding' is present, QualToQMMapping.exceeds is true; otherwise it is false. <String> represents QualToQMMapping.realWorldLabel. <QualitativeLabel> represents QualToQMMapping.qualitativeEvaluation. If <QualitativeLabel> is 'denied', 'weaklyDenied', 'weaklySatisfied', 'satisfied', 'conflict', 'unknown', or 'none', then QualToQMMapping.qualitativeEvaluation is Denied, WeaklyDenied, WeaklySatisfied, Satisfied, Conflict, Unknown, or None, respectively. <QuantitativeValue> maps to QualToQMMapping.evaluation.

### Listing B.4(j) – Grammar for Textual GRL (continued)

```
<ContributionContextGroup> ::= contributionContextGroup <Name> <colon>
    <ReferenceToContributionContext> {<comma> <ReferenceToContributionContext>}*

<ContributionContext> ::= contributionContext <Name> <left curly bracket>
    <ContributionChange>*
    [includes <ReferenceToContributionContext> {<comma>
        <ReferenceToContributionContext>}*]
<right curly bracket>

<ContributionChange> ::=
    <QualifiedReferenceToContribution> with {<ContributionType> |
        <QuantitativeValue>}
```

A <ContributionContextGroup> represents a ContributionContextGroup. <Name> is mapped to ContributionContextGroup.name according to the rules specified for <Name>. <ReferenceToContributionContext> items uniquely identify ContributionContextGroup.contribs.

A <ContributionContext> represents a ContributionContext. <Name> is mapped to ContributionContext.name according to the rules specified for <Name>. Its <ContributionChange> items represent ContributionContext.changes. <ReferenceToContributionContext> items uniquely identify ContributionContext.includedContexts.

A <ContributionChange> represents a ContributionChange. <QualifiedReferenceToContribution> uniquely identifies ContributionChange.contribution. If <ContributionType> is 'make', 'help', 'somePositive', 'unknown', 'someNegative', 'hurt', or 'break', then ContributionChange.newContribution is Make, Help, SomePositive, Unknown, SomeNegative, Hurt, or Break, respectively. <QuantitativeValue> represents ContributionChange.newQuantitativeContribution.

### Listing B.4(k) – Grammar for Textual GRL (continued)

```
<ReferenceToGRLContainableElement> ::=
    <ReferenceToIntentionalElement> | <ReferenceToIndicator>
<ReferenceToIntentionalElement> ::= <Name>
<ReferenceToIndicator> ::= <Name>
<QualifiedReferenceToIntentionalElement> ::= <QualifiedTURNIdentifier>
<QualifiedReferenceToIndicator> ::= <QualifiedTURNIdentifier>
<ReferenceToEvaluationStrategy> ::= <Name>
<ReferenceToIndicatorConversion> ::= <Name>
<ReferenceToContributionContext> ::= <Name>
<QualifiedReferenceToContribution> ::= <QualifiedTURNIdentifier>
```

The <Name> of a <ReferenceToIntentionalElement> uniquely identifies the referenced IntentionalElement. The <QualifiedTURNIdentifier> of a <QualifiedReferenceToIntentionalElement> uniquely identifies the referenced IntentionalElement. The <QualifiedTURNIdentifier> of a <QualifiedReferenceToIndicator> uniquely identifies the referenced Indicator. The <Name> of a <ReferenceToEvaluationStrategy> uniquely identifies the referenced EvaluationStrategy. The <Name> of a <ReferenceToIndicatorConversion> uniquely identifies the referenced IndicatorConversion. The <Name> of a <ReferenceToContributionContext> uniquely identifies the referenced ContributionContext. The <Name> of a <ReferenceToContribution> uniquely identifies the referenced Contribution.

### Listing B.4(l) – Grammar for Textual GRL (continued)

```
<QuantitativeValue> ::= [<plus sign> | <minus sign>]
    {<decimal digit> [<decimal digit>] | 100}
<RealWorldValue> ::= [<plus sign> | <minus sign>] <integer literal> // an Integer
```

```

<PositiveInteger> ::= <integer literal> // an Integer that is greater than 0

<Name>, <QualifiedTURNidentifier>, <String>, <Text> are defined in Listing B.1.

<plus sign> ::= +
<minus sign> ::= -

<left curly bracket>, <right curly bracket>, <colon>, <comma>, and <long arrow> are
defined in Listing B.1.

```

<decimal digit> and <integer literal> are defined in clause 9.

## B.4 Use Case Map

The grammar in this annex covers clause 8, i.e., the UCM concepts of URN. All concepts of the abstract grammar of UCM are covered by TURN except for the performance annotations in clause 8.6 (see Figure I.7), ComponentType, and EmptyPoint. Performance annotations including the probability attributes of NodeConnection and PluginBinding, as well as ComponentTypes do not constitute an essential part of the UCM notation. EmptyPoint is covered indirectly as needed for TURN.

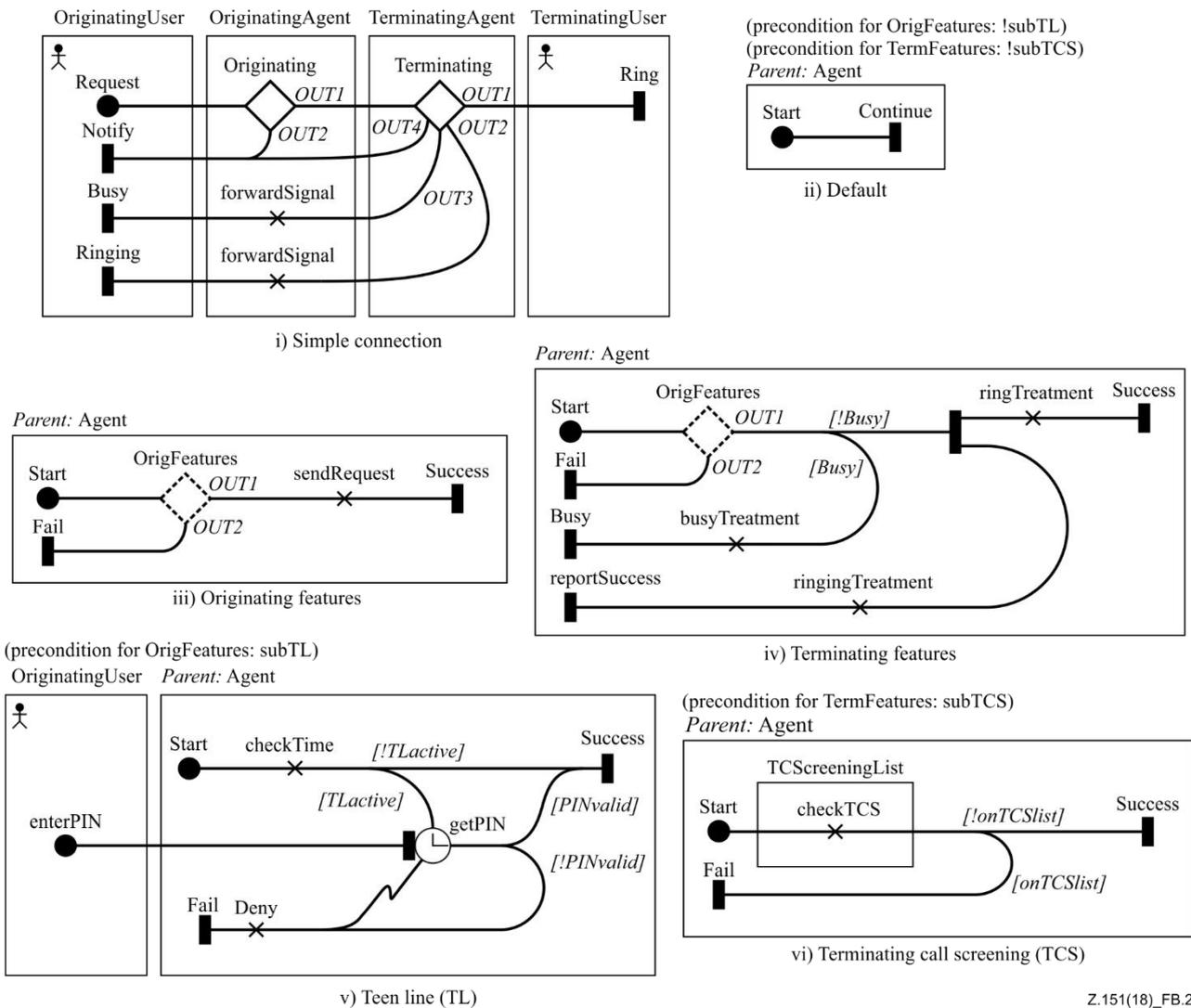


Figure B.2 – Example UCM model

While UCM does not have the explicit concept of a Path, the Textual UCM mirrors closely the abstract grammar of UCM. It specifies a UCM map as (i) a set of paths composed of path nodes and

connections between them and (ii) a set of component references which binds path nodes to components. In addition, the Textual UCM allows plugin bindings to be specified on the map which contains a corresponding plugin stub. This textual representation may also be used by the graphical representation of UCM.

Figure B.2 repeats the Example UCM model shown in Figure 61 but annotates the maps with the plugin bindings summarized informally in clause 8.2.1(e). Listings B.5(a)-(g) show a possible representation of this UCM model in the Textual UCM. The example is interspersed with explanatory text that introduces the textual UCM by example.

This example shall later be used to describe some of the alternative representations afforded by the Textual UCM, see Listing B.6 through B.9.

Listing B.10 shows the grammar of the Textual UCM except for scenario definitions along with the mapping to be applied to a specification in textual UCM to yield the abstract syntax of UCM. The description of this mapping adheres to the guide lines established in [ITU-T Z.111]. Listings B.11 and B.12 show a concrete example for the textual syntax of scenario definitions and the corresponding grammar, respectively. The example is based on the first three scenario definitions from Table 12 (see clause 8.5.2).

The representation of the UCM model depicted in Figure B.2 in Textual UCM is given in Listing B.5 below. The listing is interspersed with explanations of the preceding portion of the listing.

#### Listing B.5(a) – TURN for example UCM model from Figure B.2

```
map Default {
    start> -> continue.
}
```

The above Listing B.5(a) specifies the map "Default". The map has a single start point, "start" and a single end point "continue". A start point is indicated by its name followed by the symbol ">", that is, "start>" and is meant to indicate that the path begins here (comes out of the "start"). An end point is indicated by the name of the end point followed by the full stop (".") symbol, that is "end.". The node connection between them is indicated by the arrow ("->") symbol.

#### Listing B.5(b) – TURN for example UCM model from Figure B.2 (continued)

```
map "Simple Connection" {
    request> -> in1>Originating("Originating Features": success=out1, fail=out2,
                                start=in1, Agent=OriginatingAgent) {
        out1> -> in1>Terminating("Terminating Features": success=out1,
                                reportSuccess=out2, busy=out3, fail=out4,
                                start=in1, Agent=TerminatingAgent) {
            out1> -> ring.
            out2> -> forwardSignal -> ringing.
            out3> -> forwardSignal -> busy.
            out4> -> notify.
        }
        out2> -> notify.
    }
    actor OriginatingUser: request, notify, busy, ringing
    team OriginatingAgent: Originating, forwardSignal
    team TerminatingAgent: Terminating
    team TerminatingUser: ring
}
```

The above Listing B.5(b) specifies the map "Simple Connection". Names (e.g., the name of this map) do not need to adhere to the syntax of identifiers but can be given as a string. The map has a single start point, "request", indicated by "request>". The path leads from this start point to the stub

"Originating". The plugin bindings for this stub are given in parenthesis. This map is a static map as only one set of plugin bindings is provided. The stub is bound to the map "Originating Features". The plugin bindings describe two out-paths, labelled "out1" and "out2" and an in-path labelled "in1". The former is bound to the end point "success" on the plugin map, the latter is bound to the end point "fail". The single in-path labelled "in1" is bound to the start point "start". The plugin map also has a component binding for a component reference labelled "Agent" which is bound to the component reference "OriginatingAgent" on the calling map.

The paths enters the stub at an in-path, which is shown by its name, followed by a trailing ">" symbol, followed by the stub.

The paths exiting the stub are shown within the curly braces. There are two such paths. The path starting at out-path "out1" begins with the symbol "out1>" and leads to the stub "Terminating". The path starting at out-path "out2" begins with the symbol "out2>" and leads to the end point "notify". For the stub "Terminating", there are four out-paths, labelled "out1" through "out4". The first out-path (from "out1>") leads to the end point "ring". The second (from "out2>") leads to a responsibility named "forwardSignal" and then to the end point "ringing". The third path (from "out3>") leads to another responsibility named "forwardSignal" and then to the end point "busy". The last path (from "out4>") leads to the end point "notify".

Note that Figure B.2 shows the paths leading to end point "notify" merging at an OR-join before reaching the end point. The Textual UCM provides a convenient short cut: The OR-join may be elided and the paths may be shown as directly connecting to the end point.

After the specification of the path from "request>", the component references are defined. A component reference gives the kind of component, the name of the component reference, and then a list of the path nodes bound to that component reference. For example, the actor OriginatingUser contains the start point "request", and the end points "notify", "busy", and "ringing".

Both responsibilities named "forwardSignal" are bound to the component reference OriginatingAgent and therefore need not be distinguished.

UCM allows nodes in a specification to have the same name. The graphical syntax only supports showing that name. In order to uniquely identify a node, the textual UCM allows the id of UCM model elements to be used also. Both id and name may conform to the syntax of an identifier or they may be arbitrary text strings. When text is shown in a definition, it is the id of the node. If both id and name shall be shown, they id is shown first, followed by the hash ("#") symbol, followed by the name. For example, we could have differentiated the responsibilities as "fwd1#forwardSignal" and "fwd2#forwardSignal" and then referred to them by their id in the component reference. When a text is shown in a reference, it may be either the id or the name and must be resolved.

#### Listing B.5(c) – TURN for example UCM model from Figure B.2 (continued)

```
map Default {
    start> -> continue.
    parent Agent: start, continue
}
```

In Listing B.5(a), the component binding was omitted. In this map, the component is defined as being provided by the context. That is, when this map is used as a plugin map, a component binding must be provided. A component is specified as defined by context by using the name "parent" as component kind (see Listing B.5(c)). The actual component kind is taken from the component provided in the component binding.

#### Listing B.5(d) – TURN for example UCM model from Figure B.2 (continued)

```
map "Originating Features" {
    start> -> OrigFeatures(
```

```

    [!subTL] Default: continue=out1, start=in1, Agent=Agent
    [subTL] "Teen Line (TL)": start=in1, success=out1, fail=out2, Agent=Agent
  ) {
    out1> -> sendRequest -> success.
    out2> -> fail.
  }
  parent Agent: start..success, fail
}

```

After the start point "start>", the path enters the stub "OrigFeatures". It is not shown at which in-path the path enters the stub. TURN allows the in-path to be omitted if either there is only one in-path, or all other in-paths have already been specified explicitly.

The map "Originating Features" contains a dynamic stub "OrigFeatures". For a dynamic stub, multiple sets of plugin bindings are provided and the active plugin bindings are selected based on some condition. There are two bindings, the first is selected when the "subTL" is false, the second is selected when "subTL" is true. The syntax for conditions is as in UCM. When "subTL" is false, the map "Default" is used as plugin map, and the out-paths and component references are bound as shown. If "subTL" is true, the map "Teen Line (TL)" is used as the plugin map.

#### Listing B.5(e) – TURN for example UCM model from Figure B.2 (continued)

```

map "Terminating Features" {
  start> -> TermFeatures(
    [!subTCS] Default: continue=out1, start=in1, Agent=Agent
    [subTCS] "Terminating Call Screening (TCS)": success=out1, fail=out2
  ) {
    out1> -> {
      [!busy] -> {
        ringTreatment -> success.
        ringingTreatment -> reportSuccess.
      }
      [busy] -> busyTreatment -> busy.
    }
    out2> -> fail.
  }
  parent Agent: start..success, start..reportSuccess, start..busy, fail
}

```

The map "Terminating Features" contains an OR-fork and an AND-fork. An OR-fork is denoted by a pair of curly braces; an AND-fork is denoted by a pair of braces coupled with a vertical bar symbol (i.e., "{" and "|"); the vertical bar reminds of the bar symbol used in the graphical notation). The branches following the fork are shown within the curly braces. The forks may be named. The OR-fork, reached when the stub "TermFeatures" is exited at out-path "out1", has two branches, depending on whether "busy" is true or false. If "busy" is false, we reach the AND-fork. If "busy" is true, the responsibility "busyTreatment" is performed and then the end point "busy" is reached. The AND-fork also has two branches, leading to the end points "success" and "reportSuccess", respectively. When the stub "TermFeatures" is exited at out-path "out2", the end point "fail" is reached.

For the in-path, no binding has been provided for the TCS plugin map. There is only a single in-path, connecting to the single start point "start". In TURN, such binding can be inferred and need not be specified. Similarly, a component binding is also not specified for the TCS plugin map, as there is only one component on the parent map and one parent component on the plugin map. Again, the binding can be inferred in TURN.

The component reference selects all responsibilities on the paths between "start" and "success", between "start" and "reportSuccess", and between "start" and "busy", as well as the end point "fail".

### Listing B.5(f) – TURN for example UCM model from Figure B.2 (continued)

```
map "Teen Line (TL)" {
  start> -> checkTime -> {
    [!TLactive] -> success.
    [TLactive] -> @getPIN { J1; } -> {
      [PINvalid] -> success.
      [!PINvalid] -> >J1 -> deny -> fail.
    }
  }
  enterPIN> -> ^getPIN;
  parent Agent: start..fail, success
  actor OriginatingUser: enterPIN
}
```

The map "Teen Line (TL)" contains a timer "getPIN" (the timer is indicated by the "@" symbol, which may remind of a clock). A timer has a regular path, shown as continuing after the timer. A timer may also have a timeout path which is shown inside the curly braces following the name of the timer. When the timeout path is taken, the traversal continues with "J1". When the regular path is taken, the traversal continues with the OR-fork. The timeout trigger indicated by the "^" symbol preceding a join to the timer. On this map, when the traversal enters at the start point "enterPIN", the timer is triggered.

After the OR-fork, traversal continues to the end point "success" (if "PINvalid" is true) or it continues to responsibility "deny" and exits the map at end point "fail" (if "PINvalid" is false). Before responsibility "deny", this path leads to an OR-join "J1", indicated by the leading ">" symbol (reminiscent of the shape of the paths leading to an OR-join). The continuation of a path at another path node is indicated by this path terminating with the symbol ";", see the timeout path of the "getPin" timer which continues execution at the OR-join "J1".

### Listing B.5(g) – TURN for example UCM model from Figure B.2 (continued)

```
map "Terminating Call Screening (TCS)" {
  start> -> checkTCS -> {
    [!onTCSlist] -> success.
    [onTCSlist] -> fail.
  }
  parent Agent: start..success, fail, TCSCreeningList
  team TCSCreeningList: checkTCS
}
```

The TURN provides a number of short cuts or constructions to make modeling of large specifications more convenient. Some of these are highlighted in Listing B.5(g) and in the following examples, Listing B.6 through B.9. In map "Terminating Call Screening (TCS)" in Listing B.6(g), the binding of elements to a component uses the "." short cut notation, which only indicates the first and last element of a path segment. All path elements in this path segment are bound to the component.

### Listing B.6 – Stub declaration

```
map "Originating Features" {
  OrigFeatures = (
    [!subTL] Default: continue=out1, start=in1, Agent=Agent
    [subTL] "Teen Line (TL)": start=in1, success=out1, fail=out2, Agent=Agent
  )

  start> -> OrigFeatures {
    out1> -> sendRequest -> success.
    out2> -> fail.
  }
}
```

At times, the plugin bindings are long and visually disrupt the flow of the path. To make the path easier to read, plugin bindings may be given separately in the form of a definition, as shown in Listing B.6 for the map "Originating Features". The name of the stub is used to identify the plugin bindings. On the path, only the name of the stub is used. This form of specifying plugin bindings is also used in the graphical representation of UCM.

#### Listing B.7 – Implicit OR-join before referenced path node

```
map "Teen Line (TL)" {
  start> -> checkTime -> {
    [!TLactive] -> success.
    [TLactive] -> @getPIN { deny; } -> {
      [PINvalid] -> success.
      [!PINvalid] -> deny -> fail.
    }
  }
  enterPIN> -> ^getPIN;
}
```

In Listing B.5(f), an OR-join was shown to merge two branches before performing the responsibility "deny". As this situation is common, the TURN allows the short cut to show a path leading directly to a path node. An implicit OR-join is created in such situations. In the example above, when the timeout occurs, the path continues at the responsibility "deny".

#### Listing B.8 – Implicit OR-join at end of path

```
map "Teen Line (TL)" {
  start> -> checkTime -> {
    [!TLactive] -> success.
    [TLactive] -> @getPIN { deny; } -> {
      [PINvalid] -> success.
      [!PINvalid] -> ;
    } -> deny -> fail.
  }
  enterPIN> -> ^getPIN;
}
```

At times we may consider that the continuation of one or more branches of an OR-fork, AND-fork, or a stub is important to be highlighted, or maybe all branches join together. This situation can be depicted by a branch continuing after the closing brace of the OR-fork, AND-fork, or stub. In that case, an implicit OR-join is created (AND-join in the case of an AND-fork), and all branches that are not terminated will continue at that implicit OR-join. A branch that is not terminated is indicated by a lone ";" symbol. For a stub, when the single path implied by the plugin bindings is not shown within the braces following the stub, it is assumed to begin at the implicit OR-join after the stub. This allows for a convenient notation for stubs with a single out-binding, in which case no braces are shown and the path is shown continuing immediately after the stub.

In above example, the path is shown to continue with the responsibility "deny" after the second OR-fork. The path that is taken when "PINvalid" is false is shown to lead to this continuation.

#### Listing B.9 – Path fragments

```
map "Teen Line (TL)" {
  start> -> checkTime -> {
    [! TLactive] -> success.
    [TLactive] -> @getPIN { deny; } -> {
      [PINvalid] -> success.
      [! PINvalid] -> deny;
    }
  }
}
```

```

}
deny -> fail.
enterPIN> -> ^getPIN;
}

```

At times there are deeply nested paths where path fragments merge many times. Sometimes, in such situations, a place where a path or a group of paths merges is considered an important structuring tool to understand the behaviour of the map. In a graphical specification, we can draw attention to such path nodes by putting them in the center of the diagram or leaving space around that point on the map so that the user's attention is drawn to those nodes. In TURN, we can draw attention to such nodes by interrupting the path at such nodes and placing them at the start of a path. In the example in Listing B.9, the path fragment starting with the responsibility "deny" is considered a central point of the map. We can place it at the top level of the map, and have paths continue there from deeply nested locations. The user could read above map as: "The path leads either to success or to denial and failure."

Listing B.10 provides the grammar for the Textual UCM except for scenario definitions. The paragraphs interspersed between the listings of the grammar describe the mapping from the concrete syntax to the URN abstract syntax. The mapping is provided by describing (i) constraints which apply to a Textual UCM specification, (ii) transformations that remove concepts or syntax introduced by the Textual UCM, and (iii) a description of the relationship between the concrete syntax and the abstract syntax.

Transformations involve the concrete syntax of the Textual UCM only. The transformations are applied in the following order: (i) inline stub declarations, (ii) simplify, (iii) realize multiple start points, (iv) realize multiple end points, (v) introduce implicit joins, (vi) migrate lifted path nodes, (vii) migrate trailing path nodes, (viii) remove dangling joins, and (ix) fix bindings.

Subsequent to applying these transformations, nodes of the abstract syntax are created corresponding to the concrete syntax elements found in a Textual UCM specification.

#### Listing B.10(a) – Grammar for Textual UCM

```

<UCMspec> ::= { <UCMmap> | <ScenarioGroup> | <ScenarioDef> | <Variable> |
  <EnumerationType>}*

<UCMmap> ::= [singleton] map <Name> <left curly bracket>
  {<Path> | <ComponentRef> | <StubDeclaration> | <ResponsibilityAction>}*
<right curly bracket>

```

A <UCMspec> represents a UCMspec. <UCMmap> represents a UCMmap. <Name> is mapped to UCMmap.name according to the rules specified for <Name>. If 'singleton' is present, UCMmap.singleton is true; otherwise it is false. <Path> represents UCMmap.nodes and UCMmap.connections. <ComponentRef> maps to UCMmap.contRefs.

#### Listing B.10(b) – Grammar for Textual UCM (continued)

```

<Path> ::=
  <StartPoint> <short arrow> <PathBody>
  | <LiftedPathBodyNode> <short arrow> <PathBody>
  | <RegularEnd>

<PathBody> ::= <PathWithRegularEnd> | <PathWithReferencedEnd>

<PathBodyWithImplicitJoin> ::= <PathBody> | <PathWithConnectToImplicitJoin>

```

If an <EndPoint> is used in the <RegularEnd>, it must have a <Name>.

A <Path> with a <StartPoint> represents a StartPoint, followed by a NodeConnection, followed by the PathNodes and NodeConnections represented by <PathBody>. The <StartPoint> represents the

StartPoint. The NodeConnection is its succ. The target of the NodeConnection is the first PathNode represented by <PathBody>.

In contrast to <PathBody>, a <PathBodyWithImplicitJoin> may end with an implicit join and is only used for the <GuardedPath> of an <OrFork>, the <GuardedPath> of a <Timer>, as well as an <AndFork>, and a <StubOutPath>.

Migrate lifted path nodes: For each <OrJoin> or <AndJoin> in a <LiftedPathBodyNode>, an arbitrary <ReferencedEnd> which references this <OrJoin> or <AndJoin> is selected. This <ReferencedEnd> is replaced by the <PathBody> of the <Path> containing the <OrJoin> or <AndJoin> as <LiftedPathBodyNode>. This <Path> is then deleted from the map. If the <OrJoin> or <AndJoin> is not referenced by a <ReferencedEnd>, the <Path> is deleted from the map.

Migrate lifted path nodes: For each <Stub> in a <LiftedPathBodyNode>, an arbitrary <ReferencedEnd> containing a <ReferenceToStub> referencing this <Stub> is located such that the <InPath> on <Stub> and <ReferencedEnd> match. This <ReferencedEnd> is replaced by the <Stub>. The <Path> is then deleted from the map.

NOTE: Other transformations explained for Listing B.10(l) ensure that only <OrJoin>, <AndJoin>, or <Stub> items will remain as <LiftedPathBodyNode> items, and hence have to be migrated.

#### Listing B.10(c) – Grammar for Textual UCM (continued)

```
<StartPoint> ::=
    [<Name>] <greater than> [<Condition>]
    | <FailureKind> [<Name>] <greater than> <FailureList>

<FailureKind> ::= abort | failure

<FailureList> ::=
    <left double square bracket> <failure list> <right double square bracket>
```

A <StartPoint> without a <FailureList> represents a StartPoint with failureKind None. <Name> is mapped to StartPoint.name according to the rules specified for <Name>. If <Name> is not specified, the name is empty. The <Condition> maps to the StartPoint.precondition, if present.

A <StartPoint> with a <FailureList> maps to a StartPoint. <Name> is mapped to StartPoint.name according to the rules specified for <Name>. If <Name> is not specified, the name is empty. If the <FailureKind> is 'abort', the FailureKind is Abort. If the <FailureKind> is 'failure', the FailureKind is Failure. The nonempty list of <failure> items represents the StartPoint.failureList string.

<failure list> is defined in clause 9.

#### Listing B.10(d) – Grammar for Textual UCM (continued)

```
<PathWithRegularEnd> ::= <PathBodyNodes> <RegularEnd>
```

A <PathWithRegularEnd> represents the PathNodes and NodeConnections represented by <PathBodyNodes> and <RegularEnd>. The first PathNode represented by <RegularEnd> is the target of the last NodeConnection represented by <PathBodyNodes>.

#### Listing B.10(e) – Grammar for Textual UCM (continued)

```
<PathWithReferencedEnd> ::= <PathBodyNodes> <ReferencedEnd> <semicolon>
```

A <PathWithReferencedEnd> represents the PathNodes and NodeConnections represented by <PathBodyNodes> and <ReferencedEnd>. The first PathNode represented by <ReferencedEnd> is the target of the last NodeConnection represented by <PathBodyNodes>.

### Listing B.10(f) – Grammar for Textual UCM (continued)

```
<PathWithConnectToImplicitJoin> ::= <PathBodyNodes> <semicolon>
```

A <PathWithConnectToImplicitJoin> represents the PathNodes and NodeConnections represented by <PathBodyNodes>. The target of the last NodeConnection represented by <PathBodyNodes> is the implicit OrJoin created for the containing stub or fork, see B.10(h). If no such implicit OrJoin exists, the specification is not well-formed.

### Listing B.10(g) – Grammar for Textual UCM (continued)

```
<PathBodyNodes> ::= <PathFragment> *  
<PathFragment> ::= <PathBodyNode> <short arrow>
```

A <PathFragment> represents a PathNode or PathNodes, as represented by the <PathBodyNode> of the <PathFragment> followed by a NodeConnection as succ of the PathNode. The PathNode represented by the following <PathFragment> is the target of the NodeConnection.

### Listing B.10(h) – Grammar for Textual UCM (continued)

```
<PathBodyNode> ::=  
  <RespRef> | <OrJoin> | <AndJoin> | <WaitingPlace> | <Timer> | <FailurePoint> |  
  <OrFork> | <AndFork> | <Stub> | <Connect>
```

Introduce implicit joins: When an <OrFork> is the <PathBodyNode> of a <PathFragment> and there are any <PathWithConnectToImplicitJoin> terminating a <GuardedPath> of the <OrFork>, then an <OrJoin> is created and inserted after the <OrFork>. The <PathWithConnectToImplicitJoin> is replaced with a <ReferenceToPathBodyNode> referencing that <OrJoin>.

Migrate trailing path nodes: Otherwise, when an <OrFork> is the <PathBodyNode> of a <PathFragment>, the first <OrJoin> or <AndJoin> among the following path body nodes is located. Then, for an arbitrary <ReferencedEnd> referencing that <OrJoin> or <AndJoin>, replace it by the path body nodes following the <OrJoin> or <AndJoin> and delete the <OrJoin> or <AndJoin> and the following path body nodes from the <PathFragment>.

Introduce implicit joins: When an <AndFork> is the <PathBodyNode> of a <PathFragment> and there are any <PathWithConnectToImplicitJoin> terminating a <PathBody> of the <AndFork>, then an <AndJoin> is created and inserted after the <AndFork>. The <PathWithConnectToImplicitJoin> is replaced with a <ReferenceToPathBodyNode> referencing that <AndJoin>.

Migrate trailing path nodes: Otherwise, when an <AndFork> is the <PathBodyNode> of a <PathFragment>, the first <OrJoin> or <AndJoin> among the following path body nodes is located. Then, for an arbitrary <ReferencedEnd> referencing that <OrJoin> or <AndJoin>, replace it by the path body nodes following the <OrJoin> or <AndJoin> and delete the <OrJoin> or <AndJoin> and the following path body nodes from the <PathFragment>.

Introduce implicit joins: When a <Stub> is the <PathBodyNode> of a <PathFragment> and there are any <PathWithConnectToImplicitJoin> terminating a <StubOutPath> of the <Stub>, then an <OrJoin> is created and inserted after the <Stub>. The <PathWithConnectToImplicitJoin> is replaced with a <ReferenceToPathBodyNode> referencing that <OrJoin>.

Migrate trailing path nodes: Otherwise, when an <Stub> with a <StubOutPath> is the <PathBodyNode> of a <PathFragment>, the first <OrJoin> or <AndJoin> among the following path body nodes is located. Then, for an arbitrary <ReferencedEnd> referencing that <OrJoin> or <AndJoin>, replace it by the path body nodes following the <OrJoin> or <AndJoin> and delete the <OrJoin> or <AndJoin> and the following path body nodes from the <PathFragment>.

Introduce implicit joins: When a <Stub> without <StubOutPath> items is the <PathBodyNode> of a <PathFragment>, then a <StubOutPath> is created such that the following path body nodes are the <PathBody> of this <StubOutPath>. The <PathName> of the <StubOutPath> is inferred from the unique <OutBinding> of the <Stub>, if such is present. Otherwise, an anonymous <PathName> is created. This <StubOutPath> is used as the single <StubOutPath> of this <Stub>.

Introduce implicit joins: When a <Timer> is the <PathBodyNode> of a <PathFragment> and there is a <PathWithConnectToImplicitJoin> terminating the <GuardedPath> of the <Timer>, then an <OrJoin> is created and inserted after the <Timer>. The <PathWithConnectToImplicitJoin> is replaced with a <ReferenceToPathBodyNode> referencing that <OrJoin>.

When a <Connect> is the <PathBodyNode> of a <PathFragment>, the <PathFragment> represents an EmptyPoint point with two succ NodeConnections, where one NodeConnection has a target path node of type Connect. The other succ NodeConnection is identified by the continuing path after the <PathFragment>. This Connect models an asynchronous trigger path to the referenced node.

NOTE – After above transformations, an <OrFork>, <AndFork>, or <Stub> should no longer be a <PathBodyNode> item of the <PathBodyNodes>. If such remain, the specification is not well-formed, as these path body nodes would not be reachable.

#### Listing B.10(i) – Grammar for Textual UCM (continued)

```
<LiftedPathBodyNode> ::=
    <RespRef> | <OrJoin> | <AndJoin> | <FailurePoint> | <OrFork> | <AndFork> |
    <Stub>
```

By transformations defined in B.10(b) and (k), all <LiftedPathBodyNode> items will be removed from the <UCMmap>.

#### Listing B.10(j) – Grammar for Textual UCM (continued)

```
<RegularEnd> ::= <EndPoint> | <OrFork> | <AndFork> | <StubWithBody> | <StubAtEnd>
```

The PathNode represented by the <RegularEnd> is the target of the NodeConnection represented by the preceding node.

#### Listing B.10(k) – Grammar for Textual UCM (continued)

```
<ReferencedEnd> ::=
    <ReferenceToPathBodyNode> | <InPath> <ReferenceToStub> | <Connect>
```

When an <InPath> followed by a <ReferenceToStub> is used in a <ReferencedEnd>, the <ReferencedEnd> represents a NodeConnection that is an InBinding of the referenced stub. The target path node is uniquely identified by the <ReferenceToStub>. The <InPath> represents the in-path to the stub referred to in the InBinding of the NodeConnection.

Introduce implicit joins: When a <Stub> is referenced as a <ReferenceToPathBodyNode>, then the default or single in-path is referenced. An <OrJoin> is inserted before the referenced <Stub>. The <ReferencedEnd> is replaced by a <ReferencedEnd> that is a <ReferenceToPathBodyNode> which references the created <OrJoin>.

When a <Connect> is used in a <ReferencedEnd>, the <ReferencedEnd> represents an EndPoint with one succ NodeConnection which has a target path node of type Connect. This Connect models a synchronous trigger path to the referenced node.

Introduce implicit join: When a <ReferenceToPathBodyNode> is used in a <ReferencedEnd>, and the <ReferencedEnd> is not referencing an <OrJoin>, an <AndJoin>, a <WaitingPlace>, a <StartPoint>, or a <Timer>, an anonymous <OrJoin> is inserted before the path body node identified by the <ReferenceToPathBodyNode>. The name of the <ReferenceToPathBodyNode> is replaced by

the anonymous name. For all other <ReferenceToPathBodyNode> items that reference the same path body node, replace the name of the <ReferenceToPathBodyNode> by the anonymous name.

NOTE – By above transformations, a <Path> that is a <RegularEnd> will be converted to a path that begins with a <LiftedPathBodyNode>, and only <OrJoin>, <AndJoin>, or <Stub> items will remain as <LiftedPathBodyNode> items.

#### Listing B.10(l) – Grammar for Textual UCM (continued)

```
<InPath> ::= <PathName> <greater than>
<PathName> ::= <TURNidentifier>
```

The <PathName> of an <InPath> identifies a NodeConnection that is the stubEntry of an InBinding.

#### Listing B.10(m) – Grammar for Textual UCM (continued)

```
<RespRef> ::= <Name>
```

A <RespRef> represents a RespRef referencing a Responsibility as respDef. <Name> is mapped to RespRef.name and RespRef.respDef.name according to the rules specified for <Name>. Responsibilities (i.e., responsibility definitions) are not modeled explicitly, but any two <RespRef> items with the same <TURNidentifier> or same <DisplayName> refer to the same definition (they represent two RespRefs with the same Responsibility).

#### Listing B.10(n) – Grammar for Textual UCM (continued)

```
<OrJoin> ::= <greater than> <Name>
```

An <OrJoin> represents an OrJoin. <Name> is mapped to OrJoin.name according to the rules specified for <Name>. If <Name> is not specified, the name is empty.

Remove dangling joins: For every <OrJoin> which is not referenced by a <ReferencedEnd>, delete this <OrJoin> from the containing <Path>.

#### Listing B.10(o) – Grammar for Textual UCM (continued)

```
<AndJoin> ::= <vertical bar> <Name>
```

An <AndJoin> represents an AndJoin. <Name> is mapped to AndJoin.name according to the rules specified for <Name>. If <Name> is not specified, the name is empty.

Remove dangling joins: For every <AndJoin> which is not referenced by a <ReferencedEnd>, delete this <AndJoin> from the containing <Path>.

NOTE – While in the Recommendation an AndJoin is at times shown with an optional condition or an optional threshold for illustration purposes, these are not part of the concrete syntax of UCM.

#### Listing B.10(p) – Grammar for Textual UCM (continued)

```
<WaitingPlace> ::= [<WaitKind>] [<Name>] <greater than> [<Condition>]
<WaitKind> ::= persistent | transient
```

A <WaitingPlace> represents a WaitingPlace. <Name> is mapped to WaitingPlace.name according to the rules specified for <Name>. If <Name> is not specified, the name is empty. If the <WaitKind> is 'transient', the WaitingPlace.waitType is Transient. If the <WaitKind> is 'persistent' or does not exist, the WaitingPlace.waitType is Persistent. The optional <Condition> maps to WaitingPlace.succ.condition.

NOTE – The arrival of a synchronous or asynchronous trigger event is modeled with a second path that is connected to the WaitingPlace, see <Connect>.

#### Listing B.10(q) – Grammar for Textual UCM (continued)

```
<Timer> ::= [WaitKind] <timer> [<Name>] [<left curly bracket>  
    <GuardedPath>  
<right curly bracket>] [<Condition>]
```

A <Timer> represents a Timer. <Name> is mapped to Timer.name according to the rules specified for <Name>. If <Name> is not specified, the name is empty. If the <WaitKind> is 'transient', the waitType is Transient. If the <WaitKind> is 'persistent' or does not exist, the waitType is Persistent. If the <GuardedPath> is present, it represents the timeout path. <Condition> represents Timer.succ.condition (the Condition on the regular path). The <Condition> of the <Guard> represents Timer.timeoutPath.condition (the Condition of the timeout path).

If the <Condition> of the <Timer> is not present, Timer.succ.condition shall evaluate to false. If the <Condition> of the <Guard> of the <GuardedPath> is not present, Timer.timeoutPath.condition shall evaluate to false.

NOTE – The arrival of a synchronous or asynchronous trigger event is modeled with a second path that is connected to the Timer, see <Connect>.

#### Listing B.10(r) – Grammar for Textual UCM (continued)

```
<Connect> ::= <trigger> <ReferenceToConnectableNode>
```

A <Connect> maps to a Connect path node. The <ReferenceToConnectableNode> uniquely identifies the target PathNode of the succ NodeConnection of the Connect.

#### Listing B.10(s) – Grammar for Textual UCM (continued)

```
<FailurePoint> ::=  
    {<trigger> [<Name>] [<Condition>] <FailureLabel>  
    | {<trigger> [<Name>] <FailureLabel> [<Condition>]}}  
  
<FailureLabel> ::=  
    <left double square bracket> <failure> <right double square bracket>
```

A <FailurePoint> represents a FailurePoint. <Name> is mapped to FailurePoint.name according to the rules specified for <Name>. If <Name> is not specified, the name is empty. The <failure> represents the FailurePoint.failure string. The optional <Condition> represents the FailurePoint.succ.condition.

<failure> is defined in clause 9.

#### Listing B.10(t) – Grammar for Textual UCM (continued)

```
<EndPoint> ::= [<Name>] [<Condition>] <full stop>
```

An <EndPoint> represents an EndPoint. <Name> is mapped to EndPoint.name according to the rules specified for <Name>. If <Name> is not specified, the name is empty. The optional <Condition> represents EndPoint.postcondition.

NOTE – The optional succ NodeConnection of an EndPoint is modeled by a <Connect>.

Realize multiple end points: When several <EndPoint> items with the same <Name> occur in a map, an <OrJoin> is inserted before one of these <EndPoint> items. All other such <EndPoint> items are replaced with a <ReferenceToPathBodyNode> that references the created <OrJoin>.

### Listing B.10(u) – Grammar for Textual UCM (continued)

```
<OrFork> ::= [<Name>] <left curly bracket>  
           <GuardedPath>*  
           <right curly bracket>
```

An <OrFork> represents an OrFork. <Name> is mapped to OrFork.name according to the rules specified for <Name>. If <Name> is not specified, the name is empty. For each <GuardedPath> there exists a NodeConnection. The first PathNode represented by each <GuardedPath> is the target of the corresponding NodeConnection. The Condition represented by each <Guard>, if present, and the implied Condition represented by the <Else> are the Conditions of the corresponding NodeConnection.

Simplify: If the <OrFork> contains only one <GuardedPath>, a second <GuardedPath> is created with an <Else> and a <PathBody> which contains an <EndPoint> with an anonymous name.

### Listing B.10(v) – Grammar for Textual UCM (continued)

```
<GuardedPath> ::= [<Guard> | <Else>] <PathBodyWithImplicitJoin>
```

An <Else> may be present only if at least one other <GuardedPath> item of the containing <OrFork> has a <Guard>. The same containing <OrFork> may have at most one <Else>.

An <Else> may not be present in the <GuardedPath> of a <Timer>.

A <GuardedPath> represents the PathNodes and NodeConnections represented by the <PathBodyWithImplicitJoin>.

### Listing B.10(w) – Grammar for Textual UCM (continued)

```
<Guard> ::= <Condition> <short arrow>
```

A <Guard> represents the first NodeConnection for the branch of an <OrFork> or <Timer>. The Condition of the NodeConnection is represented by the <Condition>.

### Listing B.10(x) – Grammar for Textual UCM (continued)

```
<Else> ::= <left square bracket> else <right square bracket> <short arrow>
```

An <Else> represents the first NodeConnection for the branch of an <OrFork> and the Condition of the NodeConnection. The Condition is the negation of the disjunction of all the Conditions represented by the <Guard> elements of the <GuardedPath> elements of the containing <OrFork>.

### Listing B.10(y) – Grammar for Textual UCM (continued)

```
<AndFork> ::= [<Name>] <left curly bracket> <vertical bar>  
           <PathBodyWithImplicitJoin>*  
           <vertical bar> <right curly bracket>
```

An <AndFork> represents an AndFork. <Name> is mapped to AndFork.name according to the rules specified for <Name>. If <Name> is not specified, the name is empty. For each <PathBodyWithImplicitJoin> there exists a NodeConnection. The first PathNode represented by each <PathBodyWithImplicitJoin> is the target of the corresponding NodeConnection. An <AndFork> must have at least two <PathBodyWithImplicitJoin> items, i.e., branches.

### Listing B.10(z) – Grammar for Textual UCM (continued)

```
<Stub> ::= <StubWithBody> | <StubHead>  
  
<StubAtEnd> ::= <StubWithBody> | <StubHead> <full stop>
```

```

<StubHead> ::=
    [<InPath>] [<Name>] <StubParameters>
    | [<InPath>] <ReferenceToStubDeclaration>
    | [<InPath>] [<ReferenceToStubDeclaration>] <StubParameter>

<StubWithBody> ::= <StubHead> <left curly bracket>
    <StubOutPath>*
<right curly bracket>

<StubDeclaration> ::= <Name> <equal sign> <StubParameters>

```

A <Stub> represents a Stub. <Name> in <StubHead> and <StubDeclaration> is mapped to Stub.name according to the rules specified for <Name>. If <Name> is not specified, the name is empty. If <InPath> is present, it represents a NodeConnection. The target path node is represented by the <Stub>. The <InPath> represents the in-path to the stub referred to in the InBinding of the NodeConnection.

Simplify: If <InPath> is not present, a default <InPath> is created from the unique <InBinding> of this <Stub> which has not yet been referenced elsewhere in the specification, see B.10(ad). If there is no such unique <InBinding>, the specification is not well-formed.

Simplify: If a <Stub> with <StubParameters> does not have a <Name> and there exists a unique name in the <PluginBinding> elements of the <StubParameters>, that name is used as the <Name> of the <Stub>.

Inline stub declaration: If a <Stub> with <StubParameter> does not have a <ReferenceToStubDeclaration> and there exists a unique name in the <PluginBinding> of the <StubParameter>, that name is used as the <ReferenceToStubDeclaration> of the <Stub>.

Inline stub declaration: When a <ReferenceToStubDeclaration> is used, the <StubDeclaration> that matches the <DisplayName> of the <Name> of the <Stub> is identified and the <ReferenceToStubDeclaration> is replaced by the <Name> and <StubParameters> of the identified <StubDeclaration>. When a <StubParameter> is present, the <Binding> elements of the <PluginBinding> override the matching <Binding> of the <StubDeclaration>.

#### Listing B.10(aa) – Grammar for Textual UCM (continued)

```

<StubOutPath> ::=
    <PathName> <greater than> [<Threshold>] <short arrow>
    <PathBodyWithImplicitJoin>

<Threshold> ::= <left square bracket> <PositiveInteger> <right square bracket>

```

A <StubOutPath> represents a NodeConnection identified by <PathName>. This NodeConnection is the stubExit of the associated OutBinding. The target of this NodeConnection is the first PathNode represented by the <PathBodyWithImplicitJoin>. The <PositiveInteger> of <Threshold> represents the NodeConnection.threshold string.

#### Listing B.10(ab) – Grammar for Textual UCM (continued)

```

<StubParameter> ::= <left parenthesis> [<PluginBinding>] <right parenthesis>

<StubParameters> ::= [<StubType>] <left parenthesis>
    <PluginBinding>*
<right parenthesis>

<StubType> ::= synchronizing | blocking

```

If `<StubType>` is empty in `<StubParameters>`, and there is one or no `<PluginBinding>` element, `Stub.dynamic`, `Stub.synchronizing`, and `Stub.blocking` are false. If `<StubType>` is empty, and there is more than one `<PluginBinding>` element, `Stub.dynamic` is true. If `<StubType>` is 'synchronizing', `Stub.dynamic` and `Stub.synchronizing` are true. If `<StubType>` is 'blocking', `Stub.dynamic`, `Stub.synchronizing`, and `Stub.blocking` are true.

#### Listing B.10(ac) – Grammar for Textual UCM (continued)

```

<PluginBinding> ::=
  [<Condition>] [<Replication>]
  { <ReferenceToUCMmap> [<colon> <BindingList>] | < BindingList> }

<Replication> ::= <PositiveInteger>

<BindingList> ::= <Binding> {<comma> <Binding>}*

<Binding> ::= <InBinding> | <OutBinding> | <ComponentBinding>

```

A `<PluginBinding>` represents a `PluginBinding`. The `<Condition>`, if present, represents `PluginBinding.precondition`. The `<Replication>`, if present, represents the `PluginBinding.replicationFactor` string. `<ReferenceToUCMmap>` uniquely identifies an associated `UCMmap`, i.e., `PluginBinding.plugin`. `<BindingList>` represents associated `InBindings`, `OutBindings`, and `ComponentBindings`. `PluginBinding.in` contains the `<InBinding>` items of the `<Binding>` items in `<BindingList>`. `PluginBinding.out` contains the `<OutBinding>` items of the `<Binding>` items in `<BindingList>`. `PluginBinding.components` contains the `<ComponentBinding>` items of the `<Binding>` items in `<BindingList>`.

A `<PluginBinding>` may omit `<InBinding>` or `<OutBinding>` items for out-paths or in-paths of the stub, which will be inferred per B.10(ad) and B.10(ae). If an omitted binding cannot be inferred, the specification is incomplete.

A `<PluginBinding>` must not include `<PathName>` items in its `<InBinding>` or `<OutBinding>` items which do not occur in an `<InPath>` or `<StubOutPath>` of the `<Stub>`, respectively, with the exception of `<PathName>` items used in inferred bindings.

Simplify: The `<ReferenceToUCMmap>` may be omitted only if there is only one `<PluginBinding>` item in the `<Stub>`. If the `<ReferenceToUCMmap>` was omitted from the `<StubParameters>`, the `<Name>` of the `<Stub>` is used as the `<ReferenceToUCMmap>`.

#### Listing B.10(ad) – Grammar for Textual UCM (continued)

```

<InBinding> ::= <ReferenceToStartPoint> <equal sign> <PathName>

```

An `<InBinding>` represents an `InBinding`. `<PathName>` represents the name of the associated `NodeConnection`.

Fix bindings: If there is no `<InPath>` for a `<Stub>` and there is a single `<InBinding>` of the `<PluginBinding>` of this `<Stub>` for which no `<InPath>` exists in a `<ReferencedEnd>` with the same `<PathName>`, then a default `<InPath>` is constructed from the `<PathName>` of this `<InBinding>` and used as the `<InPath>` of the `<Stub>`.

Fix bindings: If there is no `<InPath>` for a `<Stub>` and there is a single `<StartPoint>` on the plugin map referenced by the `<PluginBinding>` of this `<Stub>` which is not referenced in an `<InBinding>` of this `<PluginBinding>`, an `<InBinding>` is created from an anonymous `<PathName>` and the unconnected `<StartPoint>`. Then a default `<InPath>` is created from the anonymous `<PathName>` and used as the `<InPath>` of the `<Stub>`.

Fix bindings: Otherwise, if there is a single `<StartPoint>` on the plugin map referenced by the `<PluginBinding>` of a `<Stub>` which is not referenced in an `<InBinding>` of the `<PluginBinding>`,

and an <InPath> exists on the <Stub> which is not referenced in an <InBinding> of this <PluginBinding>, an <InBinding> between the <StartPoint> and the <PathName> of that <InPath> is created.

Fix bindings: Otherwise, if the <InPath> of the <Stub> is bound in an <InBinding> of this <PluginBinding> but there is a unique <ReferencedEnd> containing a <ReferenceToStub> referencing this <Stub> and the <PathName> of the <InPath> of this <ReferencedEnd> is not referenced in an <InBinding> of this <PluginBinding>, then an <InBinding> is created from this <PathName> and the <StartPoint>.

Fix bindings: If there is no <InPath> for a <Stub> and there is no plugin map referenced in a <PluginBinding> or a <PathName> cannot be inferred from the <PluginBinding> as indicated above, an anonymous name is created as <PathName>. Then a default <InPath> is created from this anonymous <PathName> and used as the <InPath> of the <Stub>.

Realize multiple start points: If the same <PathName> exists in multiple <InBinding> elements of the same <PluginBinding>, a <StartPoint> with anonymous name is created for each occurrence of <ReferenceToStartPoint>. An <OrJoin> is created and inserted as the first path body node in the <Path> starting with the <StartPoint> referenced by <PathName>. A <Path> is created for each created <StartPoint> containing that <StartPoint> and a <ReferencedEnd> referencing the created <OrJoin>.

#### Listing B.10(ae) – Grammar for Textual UCM (continued)

```
<OutBinding> ::= <ReferenceToEndPoint> <equal sign> <PathName>
```

An <OutBinding> represents an OutBinding. <PathName> represents the name of the associated NodeConnection. <ReferenceToEndPoint> uniquely identifies an EndPoint.

Fix bindings: If the path continues after a <Stub> and there is an <OutBinding> among the <PluginBindings> of this <Stub> for which no <StubOutPath> exists with a matching <PathName>, a <StubOutPath> is created with the <PathName> of the <OutBinding> as its <PathName> and the path nodes after the <Stub> as the <Path> of this <StubOutPath>. The path nodes after the <Stub> are then deleted.

Fix bindings: If the path continues after a <Stub> and there is a single <EndPoint> on the plugin map referenced by the <PluginBinding> of this <Stub> which is not referenced in an <OutBinding> of this <PluginBinding>, an <OutBinding> is created for an anonymous <PathName> and the unconnected <EndPoint>. Then a <StubOutPath> is created with the anonymous <PathName> as its <PathName> and the path nodes after the <Stub> as the <Path> of this <StubOutPath>. The path nodes after the <Stub> are then deleted.

Fix bindings: If the path continues after a <Stub> and there is no plugin map referenced in a <PluginBinding> or a <PathName> cannot be inferred from the <PluginBinding> as indicated above, a <StubOutPath> is created with an anonymous <PathName> as its <PathName> and the path nodes after the <Stub> as the <Path>. The path nodes after the <Stub> are then deleted.

Fix bindings: If the path does not continue after the <Stub> and there is a single <EndPoint> on the plugin map referenced by the <PluginBinding> of a <Stub> which is not referenced in an <OutBinding> of the <PluginBinding>, and there is a <StubOutPath> with a <PathName> that is not referenced in an <OutBinding>, an <OutBinding> between the <EndPoint> and the <PathName> in the <StubOutPath> is created.

Realize multiple end points: When several <Name> items in the <OutBinding> items of a <PluginBinding> are bound to the same <PathName>, then an anonymous <OrJoin> is created as the first path body node of the <StubOutPath> with this <PathName>. Then, for all but one of the <PathName> items, a new anonymous name is created and new <PathName> items are constructed. For each such <PathName> item, a <StubOutPath> is created which has a

<ReferenceToPathBodyNode> that references the anonymous <OrJoin> as the sole path body item in its <PathBody>. Each such <StubOutPath> is inserted into the <Stub>.

#### Listing B.10(af) – Grammar for Textual UCM (continued)

```
<ComponentBinding> ::=
    <ReferenceToComponentRef> <equal sign> <ReferenceToComponentRef>
```

A <ComponentBinding> represents a ComponentBinding. The first <ReferenceToComponentRef> uniquely identifies the pluginComponent on the plugin map. The second <ReferenceToComponentRef> uniquely identifies the parentComponent on the parent map.

If only one <ComponentRef> exists on the parent map, only one <ComponentRef> for which 'parent' is present exists on the plugin map, and no <ComponentBinding> exists that references the <ComponentRef> on the plugin map, a <ComponentBinding> between the two <ComponentRef> items is inferred.

#### Listing B.10(ag) – Grammar for Textual UCM (continued)

```
<ComponentRef> ::=
    [protected] <ComponentKind> <Name> [<colon> <BoundElementList>]
    | parent <Name> [<colon> <BoundElementList>]

<ComponentKind> ::= team | object | process | agent | actor
```

A <ComponentRef> represents a ComponentRef referencing a Component as compDef. <Name> is mapped to ComponentRef.name and ComponentRef.compDef.name according to the rules specified for <Name>. If 'protected' is present, ComponentRef.compDef.protected is true; otherwise it is false. If <ComponentKind> is 'team', ComponentRef.compDef.kind is Team. If <ComponentKind> is 'object', ComponentRef.compDef.kind is Object. If <ComponentKind> is 'process', ComponentRef.compDef.kind is Process. If <ComponentKind> is 'agent', ComponentRef.compDef.kind is Agent. If <ComponentKind> is 'actor', ComponentRef.compDef.kind is Actor. If 'parent' is present, ComponentRef.compDef.context is true; otherwise it is false.

Components (i.e., component definitions) are not modeled explicitly, but any two <ComponentRef> items with the same <TURNIdentifier> or same <DisplayName> refer to the same definition (they represent two ComponentRefs with the same Component). If two such <ComponentRef> items specify different <ComponentKind>, parent flags, or protected flags, then the specification is not well-formed.

#### Listing B.10(ah) – Grammar for Textual UCM (continued)

```
<BoundElementList> ::= <BoundElement> {<comma> <BoundElement>}*

<BoundElement> ::=
    <ReferenceToStartPoint>
    | <ReferenceToPathBodyNode>
    | <ReferenceToEndPoint>
    | <ReferencedPath>
    | <ReferenceToComponentRef>

<ReferencedPath> ::=
    <ReferencedPathElement> <double full stop> <ReferencedPathElement>

<ReferencedPathElement> ::=
    <ReferenceToStartPoint>
    | <ReferenceToPathBodyNode>
    | <ReferenceToEndPoint>
```

<BoundElementList>, if present, represents ComponentRef.nodes (a list of PathNodes) and ComponentRef.children (a list of ComponentRefs) as well as ComponentRef.comp-Def.includedComponents (a list of Components) of the <ComponentRef> in which the <BoundElementList> is contained.

A <BoundElement> that is not contained in a <ReferencedPath> represents the PathNode items represented by the element referenced by the <BoundElement>.

A <ReferencedPath> represents the PathNodes that are on the path starting with the PathNode represented by the first <ReferencedPathElement> up to and including the PathNode represented by the second <ReferencedPathElement>. The path nodes represented start with the PathNode represented by the first <ReferencedPathElement> and then follow the path that the element referenced by this <ReferencedPathElement> is on or starts. Every node encountered along this path until the element referenced by the second <ReferencedPathElement> is reached is represented, including the element referenced by the second <ReferencedPathElement>. If a stub, or-fork, or and-fork is encountered, the element referenced is represented, and each out path, guarded path, or path, respectively is followed in turn. If the end of such path is reached without encountering the element referenced by the second <ReferencedPathElement>, the nodes encountered since the stub, or-fork, or and-fork, respectively, are not represented and the next out-path, guarded path, or path, respectively, is traversed.

For any <PathNode> which is not referenced by a <BoundElement> or implicitly by a <ReferencedPath> and which has an anonymous name or is not named, if this <PathNode> is on a path between two <PathNode> items which are referenced by a <BoundElement> or implicitly by a <ReferencedPath>, then it is represented by an arbitrary one of these <BoundElement> or <ReferencedPath> items.

If a <PathNode> is bound to a <ComponentRef> individually and also as part of a <ReferencedPath>, then the individual binding takes precedence.

### Listing B.10(ai) – Grammar for Textual UCM (continued)

```

<ReferenceToPathBodyNode> ::=
    <ReferenceToRespRef> | <ReferenceToOrJoin> | <ReferenceToAndJoin> |
    <ReferenceToFailurePoint> | <ReferenceToOrFork> | <ReferenceToAndFork> |
    <ReferenceToStub> | <ReferenceToWaitingPlace> | <ReferenceToTimer>

<ReferenceToConnectableNode> ::=
    <ReferenceToWaitingPlace> | <ReferenceToTimer> | <ReferenceToStartPoint>

<ReferenceToOrJoin> ::= <Name>
<ReferenceToOrFork> ::= <Name>
<ReferenceToAndJoin> ::= <Name>
<ReferenceToAndFork> ::= <Name>
<ReferenceToWaitingPlace> ::= <Name>
<ReferenceToTimer> ::= <Name>
<ReferenceToStub> ::= <Name>
<ReferenceToUCMmap> ::= <Name>
<ReferenceToStartPoint> ::= <Name>
<ReferenceToEndPoint> ::= <Name>
<ReferenceToFailurePoint> ::= <Name>
<ReferenceToRespRef> ::= <Name>
<ReferenceToComponentRef> ::= <Name>
<ReferenceToStubDeclaration> ::= <Name>

```

The <Name> of a <ReferenceToOrJoin> uniquely identifies the referenced OrJoin. The <Name> of a <ReferenceToOrFork> uniquely identifies the referenced OrFork. The <Name> of a <ReferenceToAndJoin> uniquely identifies the referenced AndJoin. The <Name> of a <Refer-

enceToAndFork> uniquely identifies the referenced AndFork. The <Name> of a <ReferenceToWaitingPlace> uniquely identifies the referenced WaitingPlace. The <Name> of a <ReferenceToTimer> uniquely identifies the referenced Timer. The <Name> of a <ReferenceToStub> uniquely identifies the referenced Stub. The <Name> of a <ReferenceToUCMmap> uniquely identifies the referenced UCMmap. The <Name> of a <ReferenceToStartPoint> uniquely identifies the referenced StartPoint. The <Name> of a <ReferenceToEndPoint> uniquely identifies the referenced EndPoint. The <Name> of a <ReferenceToFailurePoint> uniquely identifies the referenced FailurePoint. The <Name> of a <ReferenceToRespRef> uniquely identifies the referenced RespRef. The <Name> of a <ReferenceToComponentRef> uniquely identifies the referenced ComponentRef. The <Name> of a <ReferenceToStubDeclaration> uniquely identifies the referenced StubDeclaration.

#### Listing B.10(aj) – Grammar for Textual UCM (continued)

```
<ResponsibilityAction> ::= <ReferenceToRespRef> <equal sign> <left square bracket>
    <action>
<right square bracket>
```

An <action> represents the expression of the Responsibility which is the definition of the RespRef uniquely identified by <ReferenceToRespRef> (i.e., RespRef.respDef.expression).

<action> is defined in clause 9.

An example specification for <ResponsibilityAction> is provided with scenario definitions in Listing B.11.

#### Listing B.10(ak) – Grammar for Textual UCM (continued)

```
<Name> and <Condition> are defined in Listing B.1.

<PositiveInteger> is defined in Listing B.4.

<short arrow> ::= ->
<greater than> ::= >
<left double square bracket> ::= [[
<right double square bracket> ::= ]]
<semicolon> ::= ;
<vertical bar> ::= |
<timer> ::= @
<trigger> ::= ^
<left parenthesis> ::= (
<right parenthesis> ::= )
<double full stop> ::= ..

<colon>, <comma>, <left curly bracket>, <right curly bracket>, <left square
bracket>, <right square bracket>, <equal sign>, and <full stop> are defined in
Listing B.1.
```

Listing B.11 adds preconditions and postconditions to the example scenario definitions from Table 12 to fully demonstrate the grammar in Listing B.12. For the same reason, an Integer variable, Enumeration variable, an EnumerationType, and an action for a responsibility in the "Originating Features" map are specified even though they are not needed by the example scenario definitions. The Integer variable is used as a counter which is incremented by the "sendRequest" responsibility.

#### Listing B.11 – TURN for example Scenario Definitions from Table 12

```
scenarioGroup "Feature-specific UCM scenario definitions" :
    BasicCallCore, BasicCallSuccess, BasicCallBusy

scenario BasicCallCore {
```

```

    pre subOCS == false
    pre subTL == false
    post counter == 1
    initialize subOCS = false
    initialize subTL = false
    initialize subTCS = false
    initialize busy = false
    initialize counter = 0
    start SC.request
}

scenario BasicCallSuccess {
    end SC.ring, SC.ringing
    includes BasicCallCore
}

scenario BasicCallBusy {
    initialize busy = true
    end SC.busy
    includes BasicCallCore
}

bool subOCS
bool subTL
bool subTCS
bool busy
int counter
Features existingFeatures
enum Features : TeenLine, TerminatingCallScreening, OriginatingCallScreening

map "Originating Features" {
    start> -> OrigFeatures(
        [!subTL] Default: continue=out1, start=in1, Agent=Agent
        [subTL] "Teen Line (TL)": start=in1, success=out1, fail=out2, Agent=Agent
    ) {
        out1> -> sendRequest -> success.
        out2> -> fail.
    }
    parent Agent: start..success, fail
    sendRequest = [ counter = counter + 1; ]
}

```

Listing B.12 provides the grammar for the scenario definitions of the Textual UCM. The paragraphs interspersed between the listings of the grammar describe the mapping from the concrete syntax to the URN abstract syntax.

### Listing B.12(a) – Grammar for Scenario Definitions of Textual UCM

```

<ScenarioGroup> ::= scenarioGroup <Name> <colon>
    <ReferenceToScenarioDef> {<comma> <ReferenceToScenarioDef>}*

```

A <ScenarioGroup> represents a ScenarioGroup. <Name> is mapped to ScenarioGroup.name according to the rules specified for <Name>. <ReferenceToScenarioDef> items uniquely identify ScenarioGroup.scenarios.

### Listing B.12(b) – Grammar for Scenario Definitions of Textual UCM (continued)

```
<ScenarioDef> ::= scenario <Name> <left curly bracket>
  {pre <Condition>}*
  {post <Condition>}*
  {initialize <Initialization>}*
  [start <QualifiedReferenceToStartPoint> {<comma>
    <QualifiedReferenceToStartPoint>}*]
  [end <QualifiedReferenceToEndPoint> {<comma>
    <QualifiedReferenceToEndPoint>}*]
  [includes <ReferenceToScenarioDef> {<comma> <ReferenceToScenarioDef>}*]
<right curly bracket>
```

A <ScenarioDef> represents a ScenarioDef. <Name> is mapped to ScenarioDef.name according to the rules specified for <Name>. The 'pre' <Condition> items represent ScenarioDef.preconditions. The 'post' <Condition> items represent ScenarioDef.postconditions. <Initialization> items represent ScenarioDef.initializations. <QualifiedReferenceToStartPoint> items uniquely identify ScenarioDef.startPoints. <QualifiedReferenceToEndPoint> items uniquely identify ScenarioDef.endPoints. <ReferenceToScenarioDef> items uniquely identify ScenarioDef.includedScenarios.

### Listing B.12(c) – Grammar for Scenario Definitions of Textual UCM (continued)

```
<Initialization> ::=
  {<ReferenceToBooleanVariable> <equal sign> <BooleanExpression>} |
  {<ReferenceToIntegerVariable> <equal sign> <IntegerExpression>} |
  {<ReferenceToEnumVariable> <equal sign> <enumeration literal>}
```

An <Initialization> represents an Initialization. <ReferenceToBooleanVariable>, <ReferenceToIntegerVariable>, and <ReferenceToEnumVariable> uniquely identify Initialization.variable. <BooleanExpression>, <IntegerExpression>, and <enumeration literal> represent Initialization.value.

<enumeration literal> is defined in clause 9.

### Listing B.12(d) – Grammar for Scenario Definitions of Textual UCM (continued)

```
<Variable> ::= {bool | int | <ReferenceToEnumerationType>} <variable name>
```

A <Variable> represents a Variable. <variable name> represents Variable.name. 'bool' represents a Variable with type Boolean (DatatypeKind). 'int' represents a Variable with type Integer (DatatypeKind). <ReferenceToEnumerationType> uniquely identifies a Variable with type Enumeration (DatatypeKind). In this case, Variable.enumerationType is the EnumerationType uniquely identified by <ReferenceToEnumerationType>. In all other cases, Variable.enumerationType is not defined.

<variable name> is defined in clause 9.

### Listing B.12(e) – Grammar for Scenario Definitions of Textual UCM (continued)

```
<EnumerationType> ::= enum <variable name> <colon>
  <enumeration literal> {<comma> <enumeration literal>}*
```

An <EnumerationType> represents an EnumerationType. <variable name> represents EnumerationType.name. <enumeration literal> items represent EnumerationType.values.

<enumeration literal> and <variable name> are defined in clause 9.

### Listing B.12(f) – Grammar for Scenario Definitions of Textual UCM (continued)

```
<ReferenceToScenarioDef> ::= <Name>
<ReferenceToBooleanVariable> ::= <variable name>
<ReferenceToIntegerVariable> ::= <variable name>
<ReferenceToEnumVariable> ::= <variable name>
<ReferenceToEnumerationType> ::= <variable name>
<QualifiedReferenceToStartPoint> ::= <QualifiedTURNIdentifier>
<QualifiedReferenceToEndPoint> ::= <QualifiedTURNIdentifier>

<IntegerExpression> ::= <expression>

<Condition>, <Name>, <BooleanExpression>, and <QualifiedTURNIdentifier> are defined
in Listing B.1.

<colon>, <comma>, <left curly bracket>, <right curly bracket>, and <equal sign> are
defined in Listing B.1.
```

The <Name> of a <ReferenceToScenarioDef> uniquely identifies the referenced ScenarioDef. The <variable name> of a <ReferenceToBooleanVariable> uniquely identifies a referenced <Variable> whose computed type is Boolean. The <variable name> of a <ReferenceToIntegerVariable> uniquely identifies a referenced <Variable> whose computed type is Integer. The <variable name> of a <ReferenceToEnumVariable> uniquely identifies a referenced <Variable> whose computed type is an Enumeration. The <variable name> of a <ReferenceToEnumerationType> uniquely identifies the referenced <EnumerationType>. The <QualifiedTURNIdentifier> of a <QualifiedReferenceToStartPoint> uniquely identifies the referenced StartPoint. The <QualifiedTURNIdentifier> of a <QualifiedReferenceToEndPoint> uniquely identifies the referenced EndPoint.

An <IntegerExpression> is an <expression> whose computed type is Integer as defined in clause 9.3 <expression> and <variable name> are defined in clause 9.

# Appendix I

## Summary of the URN

(This appendix does not form an integral part of this Recommendation.)

### I.1 Summary of abstract metamodel

Figure I.1 shows the top level of the abstract metamodel of URN. The diagram shows the top-level elements *URNspec* and *URNmodelElement* as well as the concepts of *URNlink* and *Metadata* that can be used for both GRL and UCM models. *Concerns* are shown because they encapsulate also both GRL and UCM model elements.

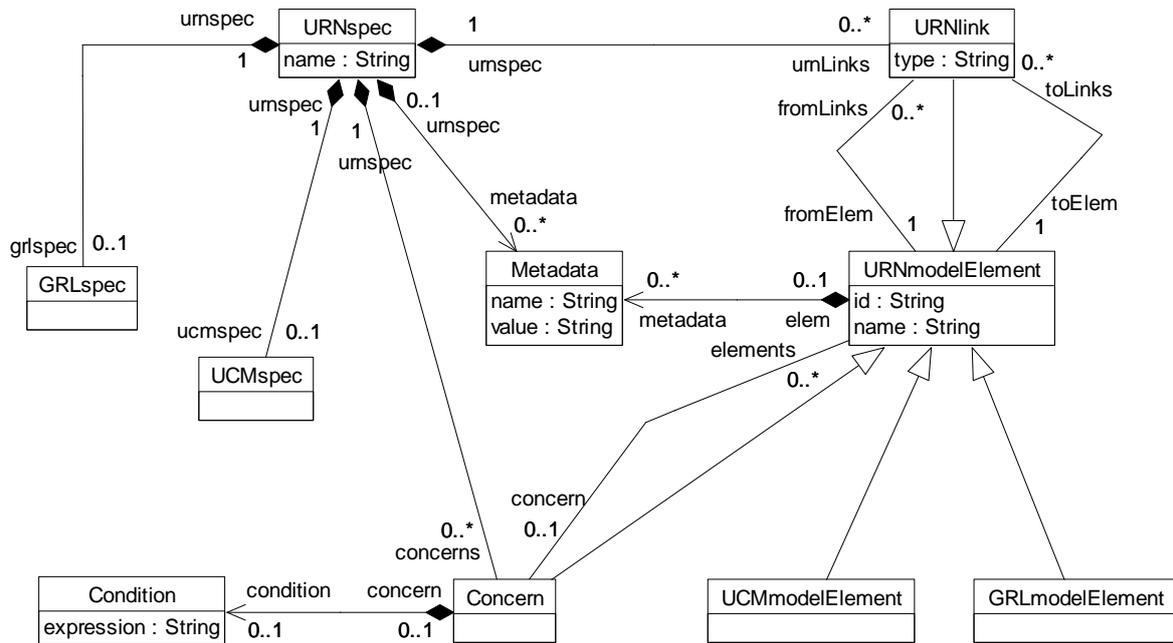
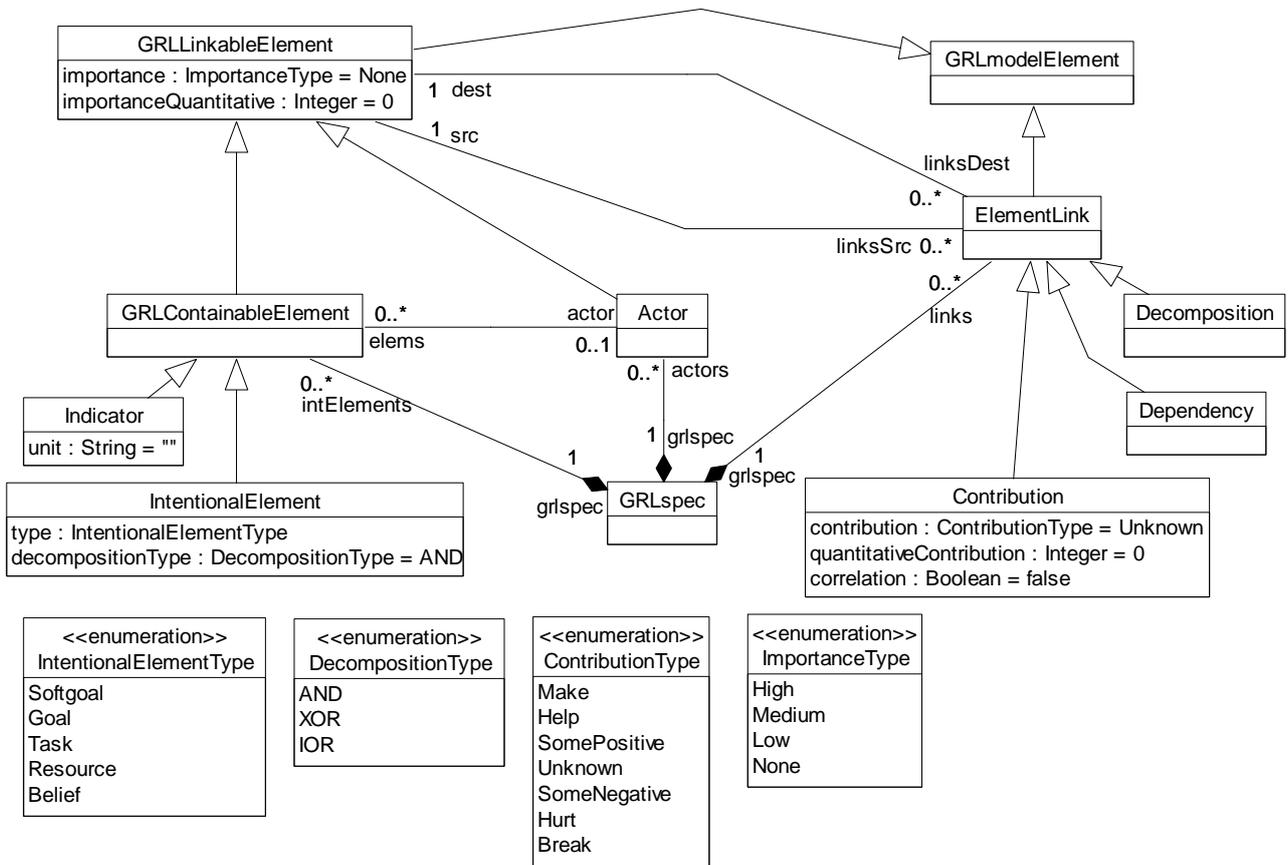


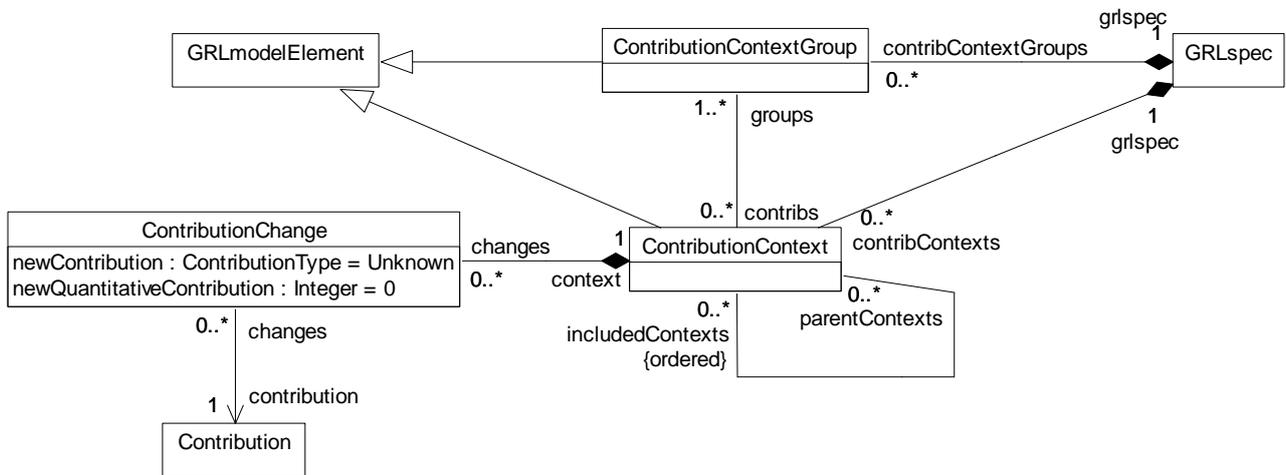
Figure I.1 – Abstract grammar: URN top level

Figure I.2 presents the core abstract metamodel of GRL. The diagram is linked to Figure I.1 via *GRLspec* and *GRLmodelElement*. The diagram shows the relationships between *ElementLinks* and *GRLLinkableElements* at the top, the relationships between *Actors*, *IntentionalElements* and *Indicators* on the middle-left and different kinds of *ElementLinks* on the middle-right.



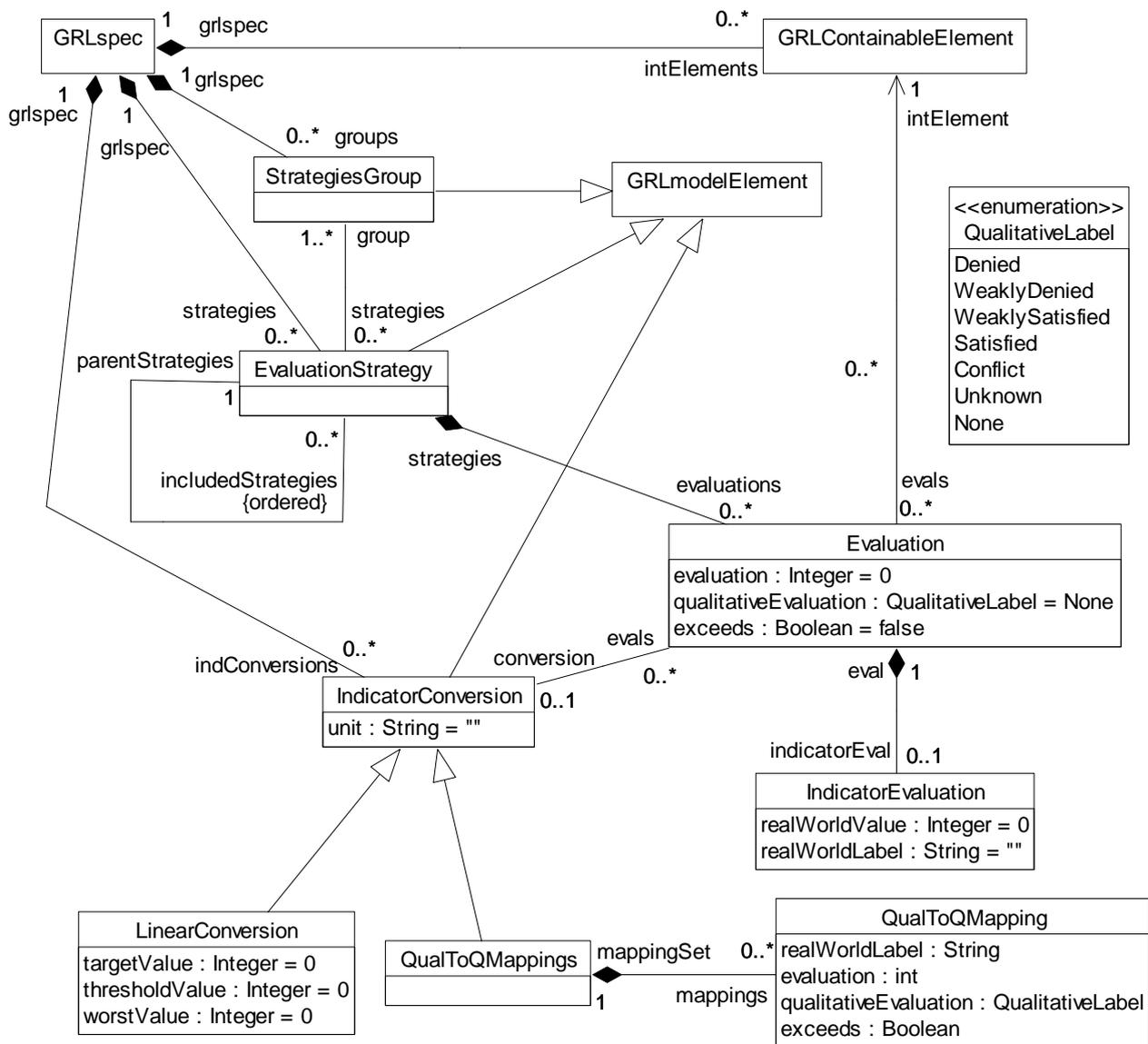
**Figure I.2 – Abstract grammar: GRL core overview**

Figure I.3 presents the abstract metamodel related to GRL *ContributionChanges* and their organization into *ContributionContexts* and *ContributionContextGroups*. The diagram is linked to Figure I.1 via *GRLspec* and *GRLmodelElement*.



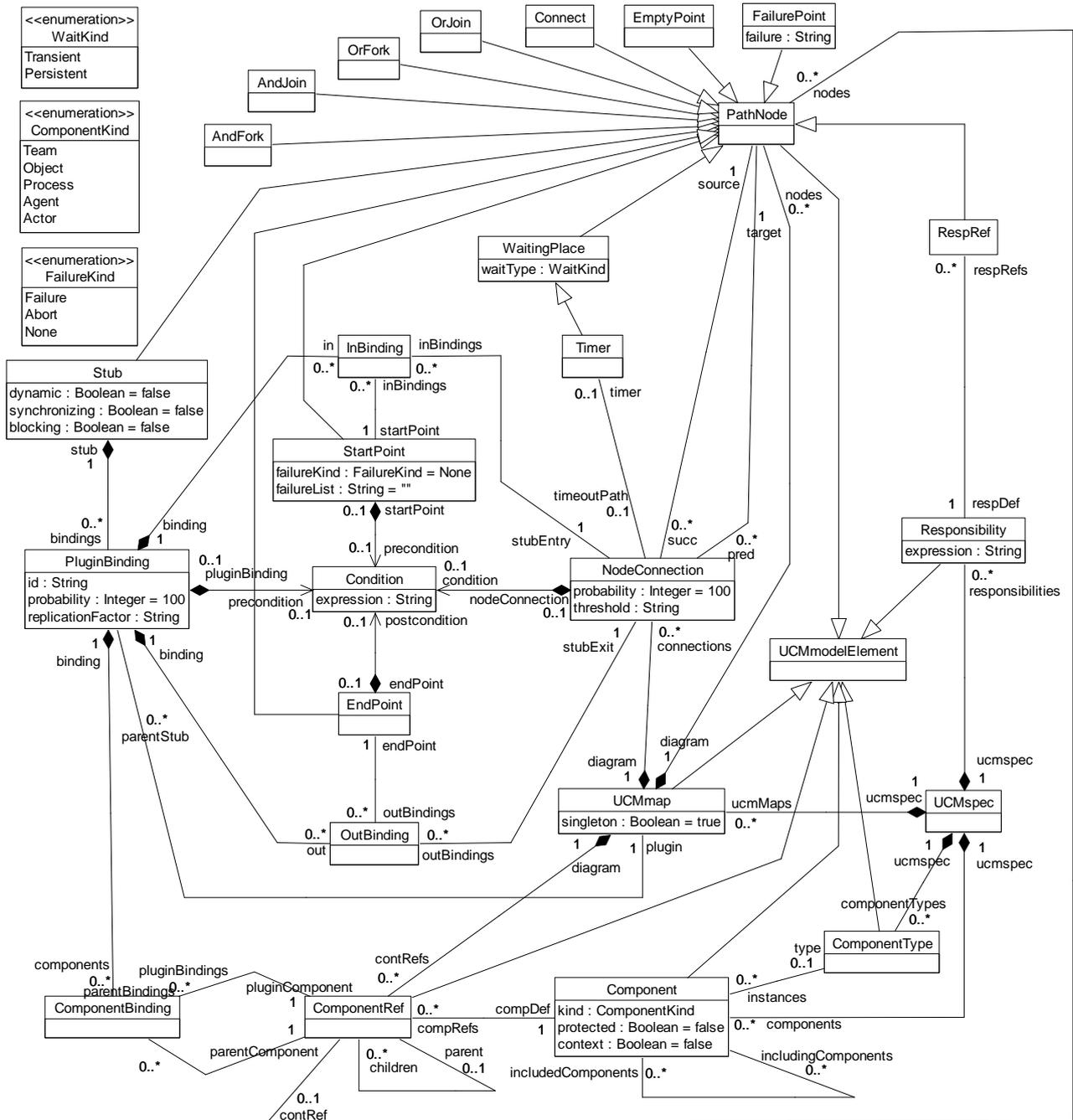
**Figure I.3 – Abstract grammar: GRL contribution changes overview**

Figure I.4 presents the abstract metamodel of GRL strategies. The diagram is linked to Figure I.1 via *GRLspec* and *GRLmodelElement*. The diagram shows concepts for the evaluation of GRL models (*StrategiesGroup*, *EvaluationStrategy* and *Evaluation*) that include indicators (*IndicatorEvaluation*, *IndicatorConversion*, *LinearConversion*, *QualToQMappings* and *QualToQMMapping*).



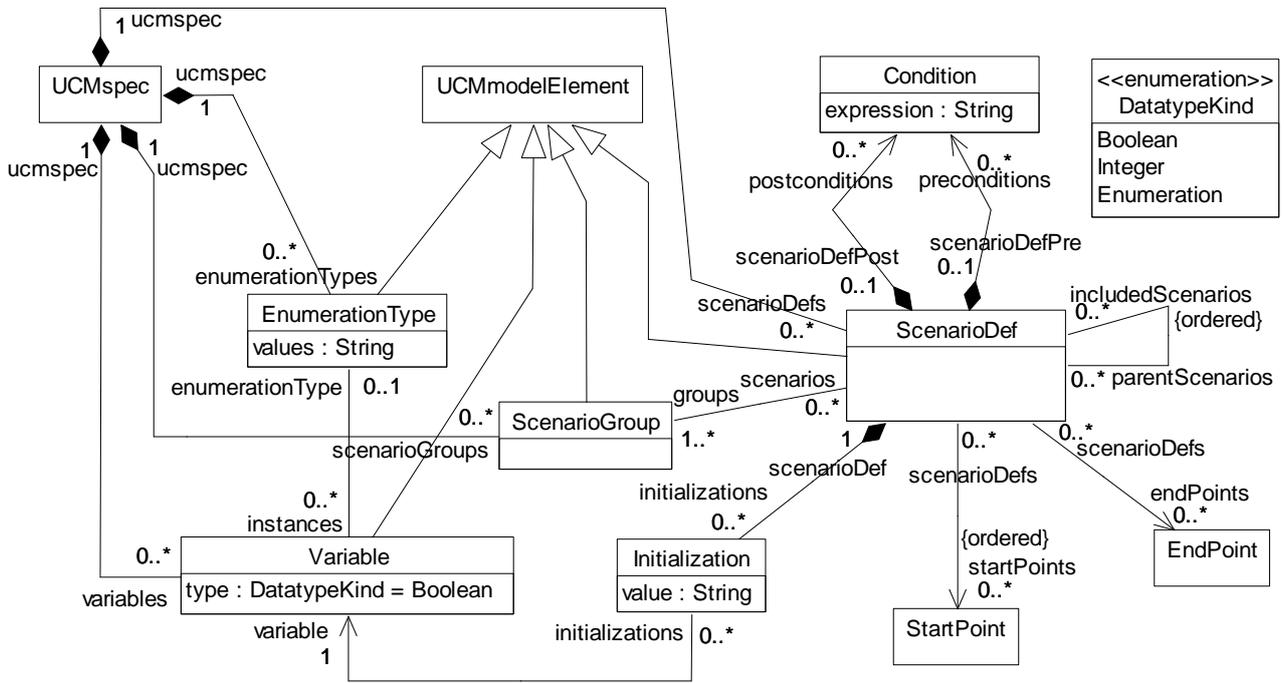
**Figure I.4 – Abstract grammar: GRL strategies overview**

Figure I.5 presents the core abstract metamodel of the UCM notation. The diagram is linked to Figure I.1 via *UCMspec* and *UCMmodelElement*. The diagram roughly shows path-related concepts at the top, plug-in binding-related concepts on the middle-left and component-related concepts at the bottom.



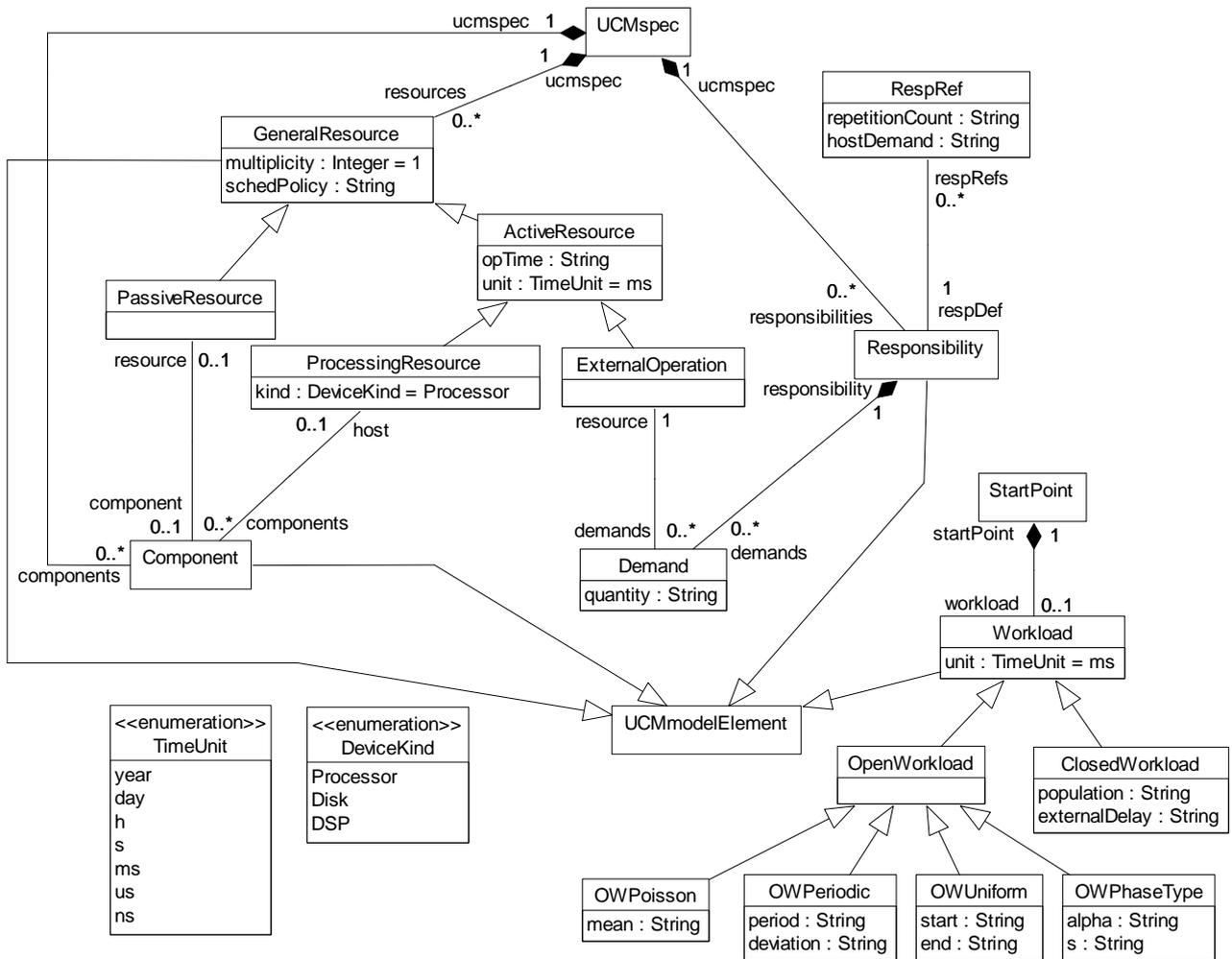
**Figure I.5 – Abstract grammar: UCM core overview**

Figure I.6 presents the abstract metamodel of UCM scenarios. The diagram is linked to Figure I.1 via *UCMspec* and *UCMmodelElement*. The diagram also introduces further relationships for *StartPoint* and *EndPoint*.



**Figure I.6 – Abstract grammar: UCM scenarios overview**

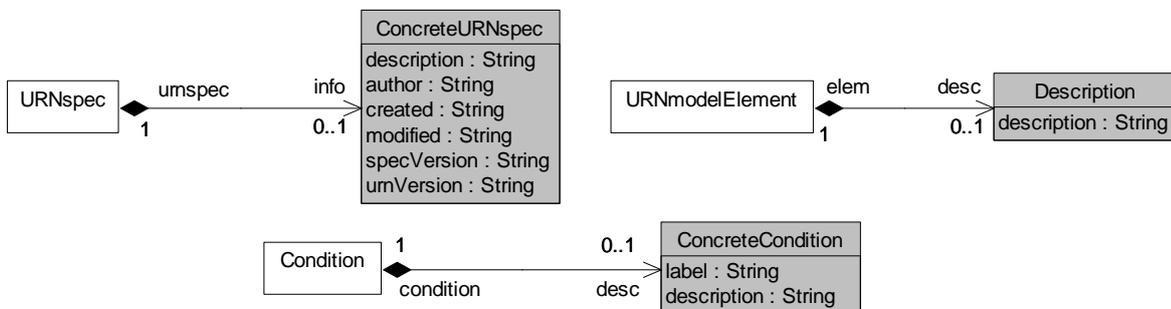
Figure I.7 presents the abstract metamodel of the performance annotations for the UCM notation. The diagram is linked to Figure I.1 via *UCMspec* and *UCMmodelElement*. The diagram also introduces further relationships for *StartPoint*, *Responsibility* and *Component*.



**Figure I.7 – Abstract grammar: UCM performance overview**

## I.2 Summary of concrete metamodel

Figure I.9 shows the top level of the concrete metamodel of URN, which extends the abstract grammar metamodel of Figure I.1. The diagram shows the top-level elements of the abstract metamodel *URNspec* and *URNmodelElement* as well as the concept of *Condition* (all defined in Figure I.1) and all their related concrete metamodel classes in grey colour.



**Figure I.8 – Concrete grammar: URN top level**

Figure I.9 presents the complete concrete metamodel of GRL, which extends the abstract grammar metamodel of Figure I.1 through Figure I.4. The diagram is linked to Figure I.1 via *GRLspec*, *URNmodelElement* and *GRLmodelElement*. It is also linked to Figure I.2 via *Actor*,



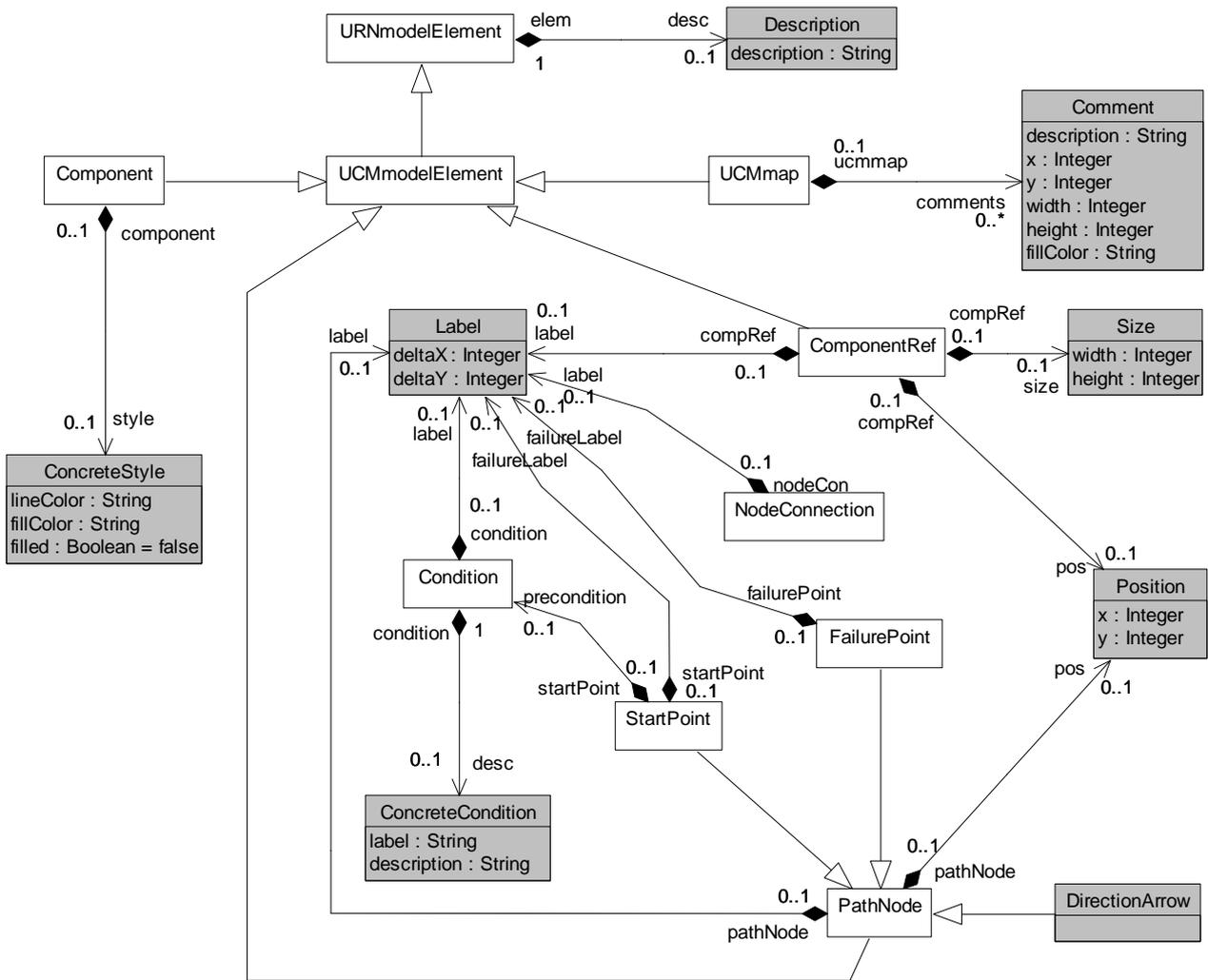
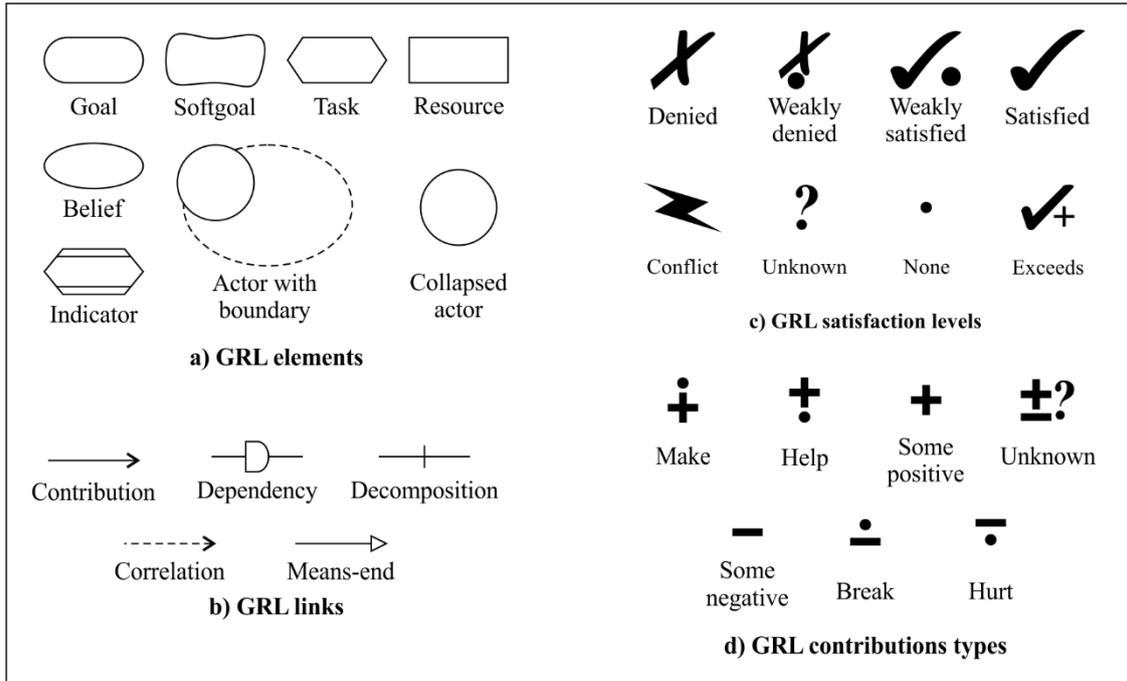


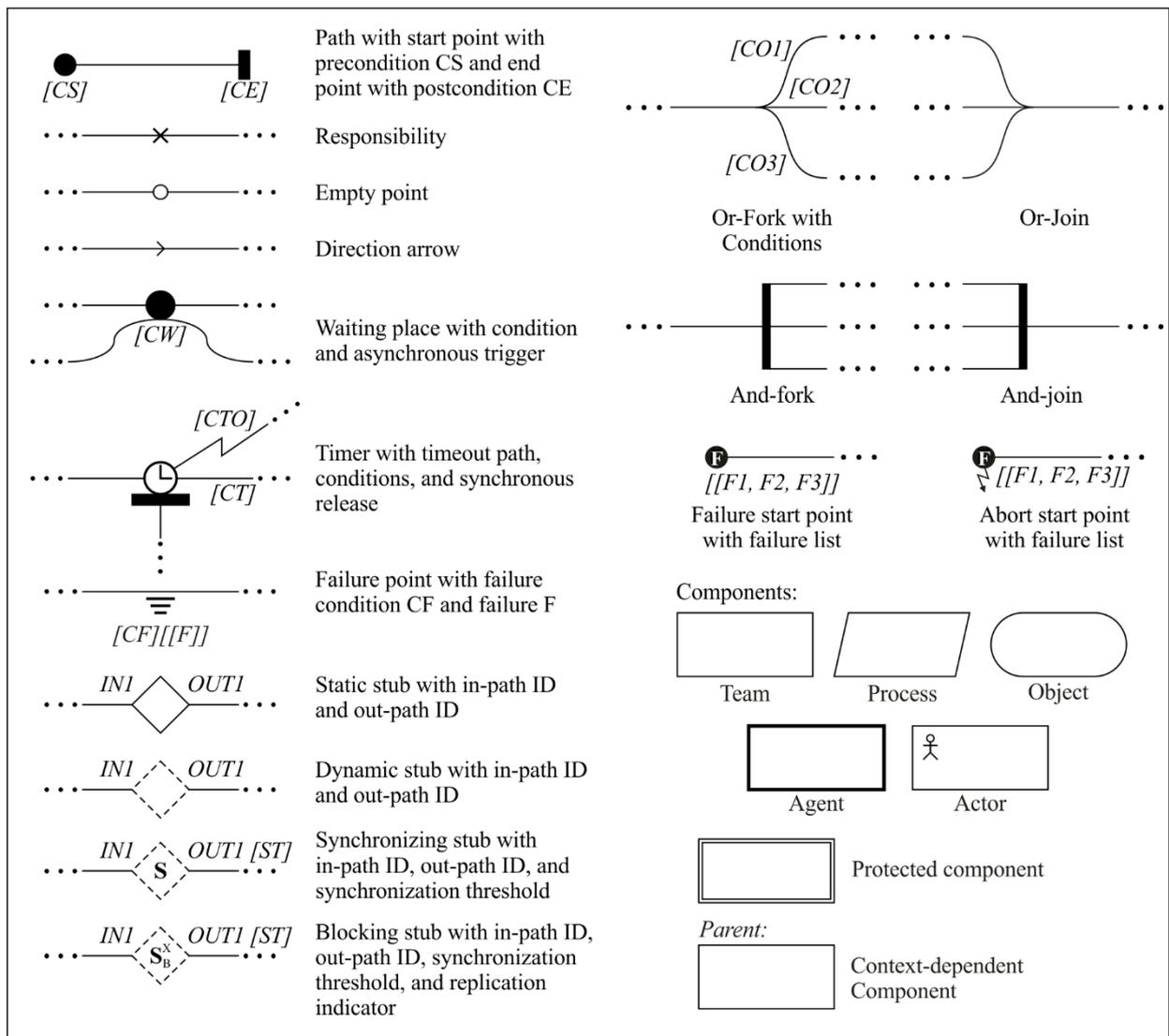
Figure I.10 – Concrete grammar: UCM

### I.3 Summary of URN symbols



Z.151(18)\_Fl.11

Figure I.11 – GRL symbols



Z.151(18)\_Fl.12

Figure I.12 – UCM symbols

## Appendix II

### Examples of GRL model evaluation algorithms

(This appendix does not form an integral part of this Recommendation.)

#### II.1 Introduction

##### II.1.1 Overview and characteristics

This appendix defines and illustrates three examples of algorithms for GRL model evaluation. These algorithms share the following common characteristics, explained in the specified numerals of clause 11.1:

- (b) Forward propagation.
- (c) Overall GRL model satisfaction is evaluated.
- (d) Actor satisfaction is evaluated.
- (e) Exceeding expectations are considered.
- (f) Fully automated.
- (g) Cycles in models are handled partially: a cycle will only be evaluated if one of its elements has a value initialized by the strategy.
- (i) Inconsistent evaluation strategies are allowed.
- (j) Evaluations defined as part of a strategy are not overridden.
- (l) Element links are evaluated in the following order: decompositions, contributions and dependencies.

A generic algorithm based on the above characteristics is presented in clause II.1.2.

The differences can be summarized as follows:

- Clause II.2: (a) Quantitative evaluation, (h) no conflict detection, (k) with relation to UCM and (n) with tolerance.
- Clause II.3: (a) Qualitative evaluation, (h) conflict detection, (k) without relation to UCM and (n) without tolerance.
- Clause II.4: (a) Hybrid evaluation, (h) no conflict detection, (k) with relation to UCM and (n) with tolerance.

As for link evaluation functions (m), they will be explained in detail for each algorithm. Algorithms are explained in plain text when they are trivial, and with pseudocode that takes advantage of the URN abstract metamodel when they are not trivial. Exceeding expectations are taken into consideration the same way for all three presented algorithms and are hence discussed in clause II.5.

##### II.1.2 Generic algorithm overview

The example algorithms all follow the same three steps: 1) initialize the evaluation values of the GRL intentional elements and indicators based on the strategy selected; 2) do a forward propagation of the evaluation values to the other elements; and 3) calculate the satisfaction of actors and the overall GRL model satisfaction. The first step follows the requirements presented in clause 11.1, and the third step depends on the type of evaluation chosen. This clause discusses the second step in more detail, as it is common to the three evaluation algorithms illustrated in this appendix.

The forward propagation algorithm in Figure II.1 follows a bottom-up, automated approach that can handle cycles partially and that does not override the initial evaluation values provided by a strategy, even when inconsistent. This algorithm takes as inputs the GRL specification and the selected strategy. It outputs a hash map containing a new evaluation value for each intentional element and

indicator. In this algorithm, each intentional element or indicator knows its number of incoming source links (*totalSourceLink*) and tracks the number of links that have been used in the propagation so far (*linkReady*).

```

Algorithm ForwardPropagation
Inputs GRLmodel:GRLspec, currentStrategy:EvaluationStrategy
Output newEvaluations:HashMap

elementsReady:List = ∅           // containable elements that can be evaluated
elementsWaiting:List = ∅       // containable elements that cannot yet be evaluated

newEvaluations = ∅

for each element:GRLContainableElement in GRLmodel.intElements
{
    element.linkReady = 0
    if (element in currentStrategy.evaluations.intElement) // is the element initialized?
        elementsReady.add(element)
    else
        elementsWaiting.add(element)
}

while (elementsReady.size() > 0)
{
    element = elementsReady.get()
    elementsReady.remove(element)
    newEvaluations.add(element, CalculateEvaluation(element, currentStrategy))
    for each link:ElementLink in element.linksSrc
    {
        destination = link.dest
        destination.linkReady = destination.linkReady + 1
        if (destination.linkReady == destination.totalSourceLink)
        {
            // all source elements have known evaluation values
            elementsWaiting.remove(destination)
            elementsReady.add(destination)
        }
    }
}

return newEvaluations

```

**Figure II.1 – Example: Forward propagation algorithm**

The forward propagation algorithm invokes the calculateEvaluation algorithm (Figure II.2), which first checks whether the element is initialized by the strategy, and if necessary computes a value from decomposition links (CalculateDecompositions), then considers contribution links (CalculateContributions) and then dependencies (CalculateDependencies). The result of CalculateDecompositions is an input for CalculateContributions, and the result of CalculateContributions is an input for CalculateDependencies. The result of CalculateDependencies is the final evaluation value. The EvaluationValue type here is a placeholder for the type of evaluation (*QualitativeLabel* for qualitative evaluations, and *Integer* for quantitative evaluations). The content of

the three sub-algorithms invoked here depends on the general type of evaluation and will be detailed for each approach.

```

Algorithm CalculateEvaluation
Inputs element:GRLContainableElement, currentStrategy:EvaluationStrategy
Output satisfactionValue:EvaluationValue

    decompValue:EvaluationValue    // intermediate result
    contribValue:EvaluationValue   // intermediate result

if not(element in currentStrategy.evaluations.intElement) // is the element not initialized?
    {
        // calculate based on decompositions, contributions, and dependencies
        decompValue = CalculateDecompositions(element)
        contribValue = CalculateContributions(element, decompValue)
        satisfactionValue = CalculateDependencies(element, contribValue)
    }
return satisfactionValue
  
```

**Figure II.2 – Example: Calculate evaluation algorithm**

## II.2 Example of quantitative evaluation algorithm

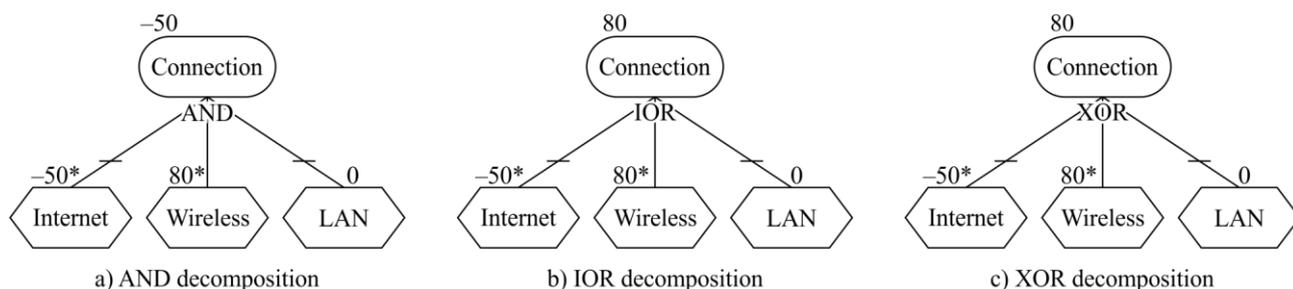
This quantitative GRL algorithm uses *Integer* values for the evaluation, and hence uses the quantitativeContribution attribute of *Contribution*, the importanceQuantitative attribute of *GRLContainableElements* and the new quantitativeVal attribute of containable elements initialized from the selected *EvaluationStrategy*.

### II.2.1 Calculating quantitative evaluations for decomposition links

This corresponds to the CalculateDecompositions(element) step in Figure II.2. The result depends on the type of decomposition (AND, IOR or XOR).

The satisfaction level of a containable element with an AND-type decomposition link is the *minimum* value of the quantitative evaluation values of its source elements. For an IOR-type decomposition link, the satisfaction level is the *maximum* value of the quantitative evaluation of its source elements. For an XOR-type decomposition link, the *maximum* is also used, but a warning is generated if more than one source element have a quantitative evaluation value different from 0.

Figure II.3 provides an example of each decomposition type based on a strategy where two sources out of three are initialized (\*). The difference between (b) and (c) here is that evaluating (c) will generate a warning as two sources have values different from 0.



Z.151(18)\_FII.3

**Figure II.3 – Example: Quantitative evaluation of decomposition links**

## II.2.2 Calculating quantitative evaluations for contribution links

This corresponds to the CalculateContributions(element, decompValue) step in Figure II.2. The total quantitative contribution is the sum of the products of the quantitative evaluation of each source element by its quantitative contribution level to the element. This value is added to decompValue up to a predefined tolerance if there is no fully satisfied or denied contribution. Correlations are treated the same way as contributions.

### Algorithm CalculateContributions

**Inputs** element:GRLContainableElement, decompValue:Integer

**Output** contribValue:Integer

```
tolerance:Integer           // predefined tolerance, between 0 and 49
oneCont:Integer            // one weighted contribution
totalCont:Integer = 0      // weighted sum of the contribution links
hasSatisfy:Boolean        // a weighted contribution of 100 is present
hasDeny:Boolean           // a weighted contribution of -100 is present
hasSatisfy = (decompValue == 100)
hasDeny = (decompValue == -100)

// compute the weighted sum of contributions
for each link:Contribution in element.linksDest
{
    oneCont = link.src.quantitativeVal × link.quantitativeContribution
    totalCont = totalCont + oneCont
    if (oneCont == 100) hasSatisfy = true
    if (oneCont == -100) hasDeny = true
}

totalCont = totalCont/100
contribValue = totalCont + decompValue

// contribution value cannot be outside [-100..100]
if (|contribValue| > 100)
    contribValue = 100 × (contribValue/|contribValue|)

// take tolerance into account if a weighted contribution of 100 or -100 is not present
if ((contribValue ≥ 100 - tolerance) and not(hasSatisfy))
    if (totalCont > 0) // positive contribution
        contribValue = max (decompValue, 100 - tolerance) // case A
        // else there is nothing to do, contribValue remains unchanged.
else if ((contribValue ≤ -100 + tolerance) and not(hasDeny))
    if (totalCont < 0) // negative contribution
        contribValue = min (decompValue, -100 + tolerance) // case B
        // else there is nothing to do, contribValue remains unchanged.

return contribValue
```

**Figure II.4 – Example: Quantitative CalculateContributions algorithm**

The algorithm in Figure II.4 ensures that the satisfaction level of each containable element will not go above 100 or below -100. In addition, the algorithm takes tolerance into account to ensure that the evaluation value of a containable element can be 100 (respectively -100) only if a) at least one of the

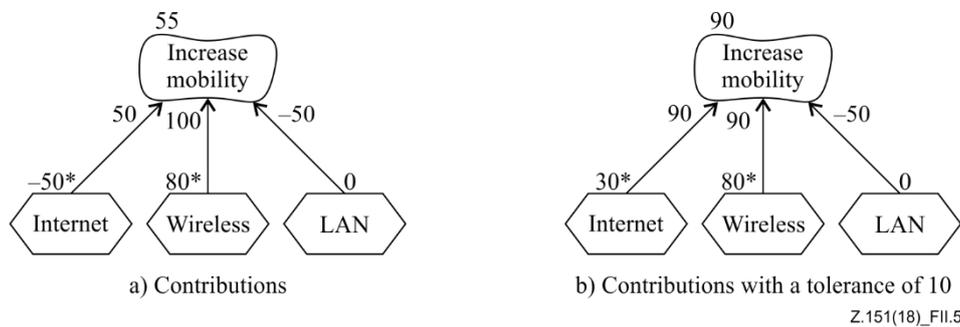
containable elements that contribute to the element has a weighted contribution of 100 (respectively  $-100$ ) or b) *decompValue* is 100 (respectively  $-100$ ). If this is not the case, then the evaluation value may be adjusted as specified in Figure II.4 and illustrated for positive values in Table II.1 (negative values are handled analogously).

NOTE – The quantitative propagation algorithm resolves conflicts.

**Table II.1 – Example: Calculating contribution values with different tolerance values**

<i>Case in Figure II.4</i>	<i>hasSatisfy</i>	<i>decompValue</i>	<i>totalCont</i>	<i>tolerance limit</i>	<i>contribValue</i>
A	false	95	3	$100 - 10 = 90$	$\max(\text{decompValue}, 90) = 95$
hasSatisfy	true			$95 + 3 = 98$	
below tolerance limit	false			$100 - 1 = 99$	$95 + 3 = 98$
B	false	95	-3	$100 - 10 = 90$	$95 - 3 = 92$
hasSatisfy	true			$95 - 3 = 92$	
below tolerance limit	false			$100 - 4 = 96$	$95 - 3 = 92$
A	false	85	13	$100 - 10 = 90$	$\max(\text{decompValue}, 90) = 90$
hasSatisfy	true			$85 + 13 = 98$	
below tolerance limit	false			$100 - 1 = 99$	$85 + 13 = 98$

Figure II.5 provides two examples with three contributions each (the initial *decompValue* is 0). Strategies initialize two elements. In (a),  $((-50 \times 50) + (80 \times 100) + (0 \times -50))/100 = 55$ . In (b), where the tolerance has been set to 10,  $((30 \times 90) + (80 \times 90) + (0 \times -50))/100 = 99$ . However, as there is no fully satisfied weighted contribution and *decompValue* is not 100, then  $100 - \text{tolerance} = 90$  is output.

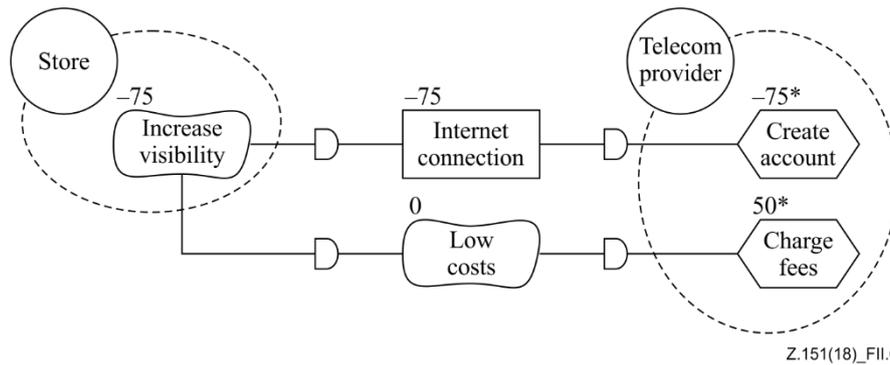


**Figure II.5 – Example: Quantitative evaluation of contribution links**

### II.2.3 Calculating quantitative evaluations for dependency links

This corresponds to the *CalculateDependencies(element, contribValue)* step in Figure II.2. In this algorithm, the source element of the dependency links cannot have an evaluation value higher than those of the containable elements it depends on (i.e., the target elements of the dependency links). This algorithm hence simply returns the minimum between *contribValue* and the evaluation values of the target elements.

A simple example is shown in Figure II.6, with a strategy that initializes the two tasks. Consequently, the qualitative values of other elements are initially set to 0. Internet Connection becomes  $-75$  since this value is less than 0. Low Costs, on the other hand, will keep its value of 0 because it is less than 50. The Increase Visibility softgoal gets the value  $\min(0, \min(-75, 0)) = -75$ .



**Figure II.6 – Example: Quantitative evaluation of dependency links**

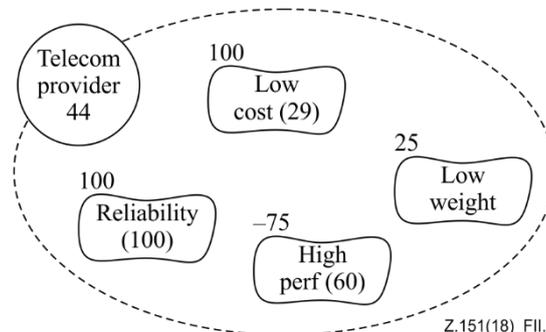
## II.2.4 Calculating quantitative evaluations for actors and the overall GRL model

This is the third and last step discussed in clause II.1.2. In order to compute the quantitative evaluation value of an actor and the overall GRL model, it is necessary to first identify the quantitative satisfaction value and quantitative importance value of each containable element bound to the actor. Only elements with an importance greater than 0 are counted (assume their number to be  $n$  and their references to be  $elem_i$  with  $i = 1..n$ ). This algorithm then computes the quantitative evaluation value of the actor as follows:

$$actor.quantitativeVal = \left( \sum_{i=1}^n elem_i.quantitativeVal \times elem_i.importanceQuantitative \right) / \sum_{i=1}^n elem_i.importanceQuantitative$$

For example, Figure II.7 shows an actor with four softgoals, three of which with non-zero importance. The quantitative value of the actor's satisfaction becomes:

$$((100 \times 100) + (100 \times 29) + (-75 \times 60)) / (100 + 29 + 60) = 44$$



**Figure II.7 – Example: Quantitative evaluation of actors**

Second, the quantitative evaluation value of the overall GRL model is calculated the same way as the quantitative evaluation value of actors except that the quantitative evaluation values and quantitative importance values of actors are used instead of containable elements. Assume that the number of actors with importance values greater than 0 are  $n$  and their references to be  $actor_i$  with  $i = 1..n$ ). This algorithm then computes the quantitative evaluation value of the overall GRL model  $m$  as follows:

$$m.quantitativeVal = \left( \sum_{i=1}^n actor_i.quantitativeVal \times actor_i.importanceQuantitative \right) / \sum_{i=1}^n actor_i.importanceQuantitative$$

## II.3 Example of qualitative evaluation algorithm

This qualitative GRL algorithm uses *QualitativeLabel* values for the evaluation, and hence uses the qualitative contribution attribute of *Contribution*, the importance attribute of *GRLContainableElements* and the new attribute *qualitativeVal* of containable elements initialized from the selected *EvaluationStrategy*. The qualitative contributions (see clause 7.4.3) are *Make*, *SomePositive*, *Help*, *Unknown*, *Hurt*, *SomeNegative* and *Break*. The qualitative evaluation labels (see clause 7.5.4) are *Satisfied*, *WeaklySatisfied*, *None*, *WeaklyDenied*, *Denied*, *Conflict* and *Unknown*. The qualitative importance values (see clause 7.1.4) are *High*, *Medium*, *Low* and *None*. Since these values are discrete, the propagation algorithm considers them individually. To this end, lookup tables and partial orderings are often used to define necessary functions explicitly.

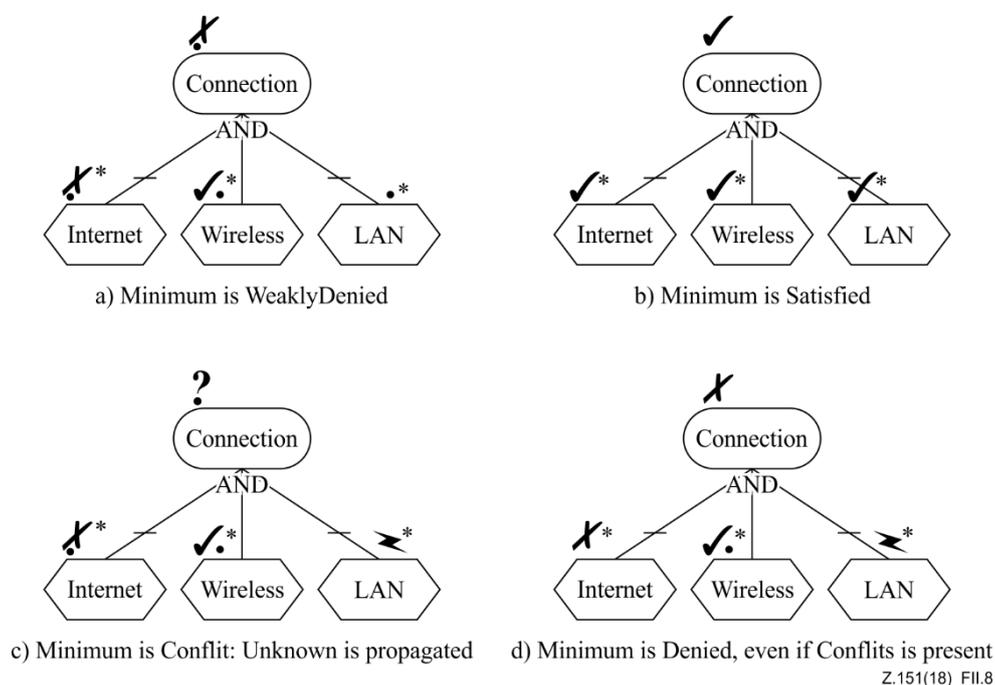
### II.3.1 Calculating qualitative evaluations for decomposition links

This corresponds to the CalculateDecompositions(element) step in Figure II.2. The result depends on the type of decomposition (AND, IOR or XOR).

The satisfaction level of a containable element with an AND-type decomposition link is the *minimum* value of the qualitative evaluation values of its source elements, where qualitative values are ordered from minimum to maximum as follows:

*Denied* < (*Conflict* = *Unknown*) < *WeaklyDenied* < *None* < *WeaklySatisfied* < *Satisfied*

However, *Conflict* results are substituted with *Unknown* as conflicts are not propagated. This simplifies the discovery of root causes (the first conflict) during the analysis of complex models. Figure II.8 provides four examples of qualitative AND-type decomposition that illustrate this propagation.

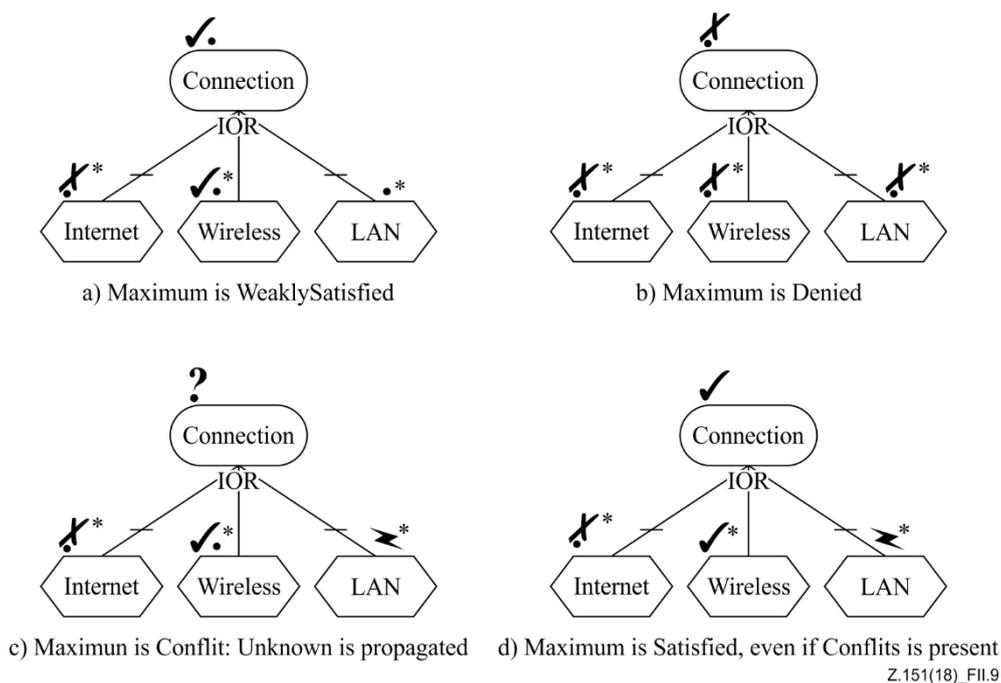


**Figure II.8 – Example: Qualitative evaluation of AND-type decomposition links**

For an IOR-type decomposition, the satisfaction level is the *maximum* value of the qualitative evaluation of its source elements, where qualitative values are ordered from minimum to maximum as follows:

*Denied* < *WeaklyDenied* < *None* < *WeaklySatisfied* < (*Conflict* = *Unknown*) < *Satisfied*

Again, *Conflict* results are substituted with *Unknown* as conflicts are not propagated. Figure II.9 provides four examples of qualitative IOR-type decomposition that illustrate this propagation.



**Figure II.9 – Example: Qualitative evaluation of IOR-type decomposition links**

For an XOR-type decomposition link, the *maximum* is propagated in the same way as for an IOR-type decomposition, but a warning is generated if more than one source element have a quantitative evaluation value different from *None*.

### II.3.2 Calculating qualitative evaluations for contribution links

This corresponds to the CalculateContributions(element, decompValue) step in Figure II.2. Correlations are treated the same way as contributions. However, unlike the quantitative evaluation of contribution links, there is no notion of tolerance here. The algorithm is presented in Figure II.10.

```

Algorithm CalculateContributions
Inputs element:GRLContainableElement, decompValue:QualitativeLabel
Output contribValue:QualitativeLabel
    oneCont:QualitativeLabel           // one weighted contribution
    ns:Integer = 0                     // number of Satisfied weighted contributions
    nws:Integer = 0                    // number of WeaklySatisfied weighted contributions
    nwd:Integer = 0                    // number of WeaklyDenied weighted contributions
    nd:Integer = 0                     // number of Denied weighted contributions
    nu:Integer = 0                     // number of Unknown weighted contributions
    weightSD:QualitativeLabel          // partial weighted contribution from ns and nd
    weightWSWD:QualitativeLabel        // partial weighted contribution from nws and nwd

// adjust the weighted contribution counters according to decompValue
    AdjustContributionCounters(decompValue, ns, nws, nwd, nd, nu)

// compute the numbers of weighted contributions for each kind
for each link:Contribution in element.linksDest
{
    oneCont = WeightedContribution(link.src.qualitativeVal, link.contribution)
    AdjustContributionCounters(oneCont, ns, nws, nwd, nd, nu)
}

// check for the presence of unknown weighted contributions
if (nu > 0)
    contribValue = Unknown
else
{
    weightSD = CompareSatisfiedAndDenied (ns, nd)
    weightWSWD = CompareWSandWD (nws, nwd)
    contribValue = CombineContributions (weightSD, weightWSWD)
}

return contribValue

```

**Figure II.10 – Example: Qualitative CalculateContributions algorithm**

The AdjustContributionCounters algorithm (Figure II.11) is first invoked by CalculateContributions to increment the weighted contribution counter that corresponds to decompValue. It is then invoked in the *for* loop to increment the counters for each individual weighted contribution computed from contribution links.

```

Algorithm AdjustContributionCounters
Inputs qualValue:QualitativeLabel
Modifies ns, nws, nwd, nd, nu:Integer

    case qualValue of
    Satisfied:           ns++
    WeaklySatisfied:    nws++
    WeaklyDenied:       nwd++
    Denied:              nd++
    Unknown:             nu++

```

**Figure II.11 – Example: AdjustContributionCounters algorithm**

The CalculateContributions algorithm also uses a WeightedContribution function that computes one qualitative weighted contribution according to the lookup table in Table II.2, where the rows specify the possible qualitative evaluation values of the source and where the columns specify the possible qualitative contribution types of the element's incoming contribution link. Note that previously found conflicts are not propagated by this function, which propagates *Unknown* instead.

**Table II.2 – WeightedContribution function for the computation of one weighted contribution**

	<i>Make</i>	<i>Help</i>	<i>SomePositive</i>	<i>Unknown</i>	<i>SomeNegative</i>	<i>Hurt</i>	<i>Break</i>
<i>Denied</i>	Denied	WeaklyDenied	WeaklyDenied	None	WeaklySatisfied	WeaklySatisfied	Satisfied
<i>Weakly Denied</i>	WeaklyDenied	WeaklyDenied	WeaklyDenied	None	WeaklySatisfied	WeaklySatisfied	WeaklySatisfied
<i>Weakly Satisfied</i>	WeaklySatisfied	WeaklySatisfied	WeaklySatisfied	None	WeaklyDenied	WeaklyDenied	WeaklyDenied
<i>Satisfied</i>	Satisfied	WeaklySatisfied	WeaklySatisfied	None	WeaklyDenied	WeaklyDenied	Denied
<i>Conflict</i>	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown
<i>Unknown</i>	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown
<i>None</i>	None	None	None	None	None	None	None

If there is at least one *Unknown* weighted contribution detected, then the result is *Unknown*. Otherwise, three functions will be used in sequence to compute the result.

The CompareSatisfiedAndDenied function determines if there are *Satisfied* values without *Denied* values, or the opposite. If none of these values are present, then *None* is returned. However, if there is at least one of each, then *Conflict* is returned. Formally:

$$\begin{aligned}
 \text{CompareSatisfiedAndDenied}(ns, nd) &= \text{Conflict, if } (ns > 0 \text{ and } nd > 0) \\
 &= \text{Satisfied, if } (ns > 0 \text{ and } nd = 0) \\
 &= \text{Denied, if } (nd > 0 \text{ and } ns = 0) \\
 &= \text{None, if } (ns = 0 \text{ and } nd = 0)
 \end{aligned}$$

The CompareWSandWD function determines if there are more *WeaklySatisfied* values than *WeaklyDenied* values, or the opposite. If their numbers are equal, then these contributions cancel each other out and *None* is returned. Formally:

$$\begin{aligned}
 \text{CompareWSandWD}(ws, wd) &= \text{WeaklySatisfied, if } (nws > nwd) \\
 &= \text{WeaklyDenied, if } (nwd > nws) \\
 &= \text{None, if } (nwd = nws)
 \end{aligned}$$

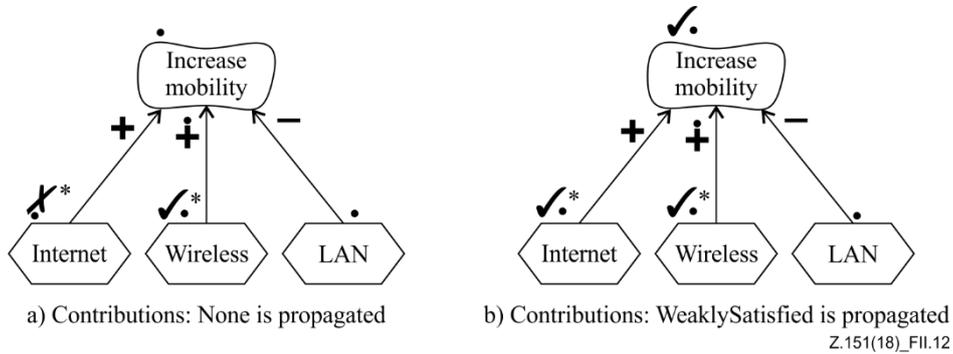
The final result is computed with the CombineContributions function, which combines the previously computed values according to Table II.3. In this table, the rows specify the possible qualitative values representing the global influence of weak contributions (i.e., weightWSWD), whereas the columns specify the possible qualitative values representing the global influence of *Satisfied* and *Denied* contributions (i.e., weightSD).

**Table II.3 – CombineContributions function for the computation of the final contribution**

	<i>Denied</i>	<i>Satisfied</i>	<i>Conflict</i>	<i>None</i>
<i>Weakly Denied</i>	Denied	Weakly Satisfied	Conflict	Weakly Denied
<i>Weakly Satisfied</i>	Weakly Denied	Satisfied	Conflict	Weakly Satisfied
<i>None</i>	Denied	Satisfied	Conflict	None

Figure II.12 provides two examples with three contributions each (the initial decompValue is *None*). Strategies initialize two elements in each example. In (a), (*WeaklyDenied* × *SomePositive*) =

*WeaklyDenied*, (*WeaklySatisfied* × *Make*) = *WeaklySatisfied*, and (*None* × *SomeNegative*) = *None*. The comparison of *Satisfied* and *Denied* results in a 0:0 tie and therefore *None*. The comparison of *WeaklySatisfied* and *WeaklyDenied* results in a 1:1 tie and therefore *None*. Finally, the combined contribution of *None* and *None* results in *None*. In (b), (*WeaklySatisfied* × *SomePositive*) = *WeaklySatisfied*, (*WeaklySatisfied* × *Make*) = *WeaklySatisfied*, and (*None* × *SomeNegative*) = *None*. The comparison of *Satisfied* and *Denied* results in a 0:0 tie and therefore *None*. The comparison of *WeaklySatisfied* and *WeaklyDenied* results in a 2:0 win and therefore *WeaklySatisfied*. Finally, the combined contribution of *None* and *WeaklySatisfied* results in *WeaklySatisfied*.



**Figure II.12 – Example: Qualitative evaluation of contribution links**

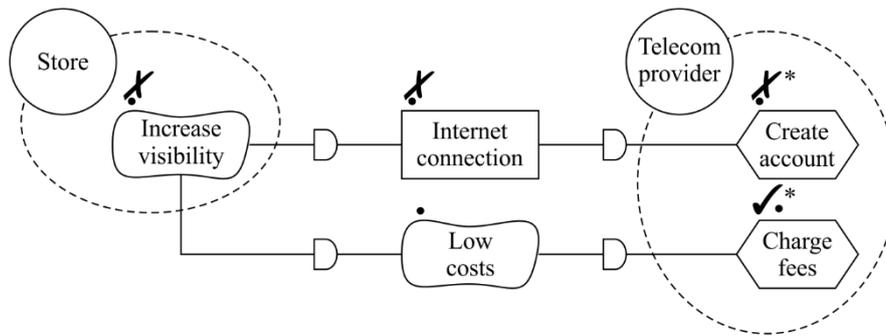
### II.3.3 Calculating qualitative evaluations for dependency links

This corresponds to the CalculateDependencies(element, contribValue) step in Figure II.2. In this algorithm, the source element of the dependency links cannot have an evaluation value higher than those of the containable elements it depends on (i.e., the target elements of the dependency links). This algorithm hence simply returns the *minimum* value between contribValue and the qualitative evaluation values of the target elements. The qualitative values are ordered from minimum to maximum in the same way as for qualitative AND-type decompositions:

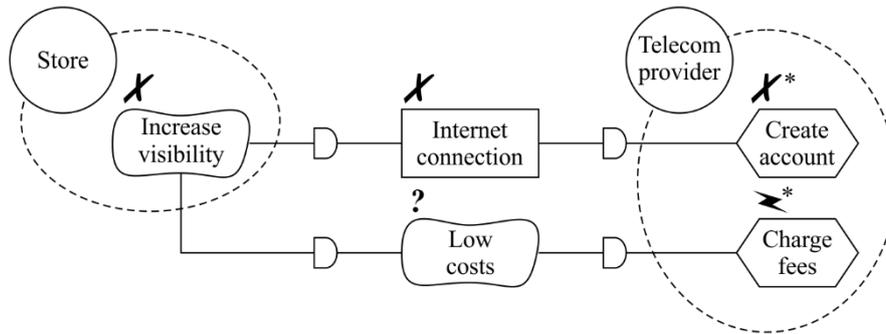
*Denied* < (*Conflict* = *Unknown*) < *WeaklyDenied* < *None* < *WeaklySatisfied* < *Satisfied*

Again, *Conflict* results are substituted with *Unknown* as conflicts are not propagated.

Two examples are shown in Figure II.13, with strategies that initialize the two tasks. Consequently, the qualitative values of other elements are initially set to *None*. Example (a) is similar to the one from Figure II.6. Internet Connection becomes *WeaklyDenied* since this value is less than *None*. Low Costs, on the other hand, will keep its value of *None* because it is less than *WeaklySatisfied*. The Increase Visibility softgoal gets the value *WeaklyDenied* because this is the minimum between *None* and *WeaklyDenied*. Example (b) illustrates that a *Conflict* value in a target element propagates to an *Unknown* value in the source element (e.g., Low Cost), unless there is a *Denied* value in another target element or in contribValue (in which case the propagated value is *Denied*, e.g., for Increase Visibility).



a) Minimum is WeaklyDenied



d) Minimum is Denied, even if Conflit is present

Z.151(18)\_FII.13

**Figure II.13 – Example: Qualitative evaluation of dependency links**

### II.3.4 Calculating qualitative evaluations for actors and the overall GRL model

This is the third and last step discussed in clause II.1.2. In order to compute the qualitative evaluation value of an actor and the overall GRL model, the qualitative satisfaction value and qualitative importance value of each containable element bound to the actor are first used.

The CalculateActorEvaluation algorithm is similar to the qualitative CalculateContributions algorithm (Figure II.10) and reuses some of its sub-algorithms. First, the qualitative evaluation value of each containable element bound to the actor is weighted according to the importance of that element to the actor. This WeightedImportance function is defined in Table II.4, where the rows specify the possible qualitative importance values of the element and where the columns specify the possible qualitative evaluation values of the element. The AdjustEvaluationCounters function is similar to the AdjustContributionCounters function (see Figure II.11) but also increments the nc counter if a *Conflict* is provided as a qualitative value input. Then, the qualitative evaluation value of the actor is calculated with the same sub-algorithms used to combine the qualitative weighted evaluation values for contribution links (i.e., CompareSatisfiedAndDenied, CompareWSandWD CombineContributions).

```

Algorithm CalculateActorEvaluation
Inputs actor:Actor
Output actorEvalValue:QualitativeLabel

    oneElemVal:QualitativeLabel    // one element value weighted according to its
importance
    ns:Integer = 0                  // number of Satisfied weighted values
    nws:Integer = 0                 // number of WeaklySatisfied weighted values
    nwd:Integer = 0                 // number of WeaklyDenied weighted values
    nd:Integer = 0                  // number of Denied weighted values
    nu:Integer = 0                  // number of Unknown weighted values
    nc:Integer = 0                  // number of Conflict weighted values
    weightSD:QualitativeLabel      // partial weighted values from ns and nd
    weightWSWD:QualitativeLabel    // partial weighted values from nws and nwd

// compute the numbers of weighted contributions for each kind
for each boundElem:GRLContainableElement in actor.elems
{
    oneElemVal= WeightedImportance(boundElem.qualitativeVal, boundElem.importance)
    AdjustEvaluationCounters(oneElemVal, ns, nws, nwd, nd, nu, nc)
}

// check for the presence of unknown and conflict weighted evaluation values
if (nc > 0)
    actorEvalValue = Conflict
else if (nu > 0)
    actorEvalValue = Unknown
else
{
    weightSD = CompareSatisfiedAndDenied (ns, nd)
    weightWSWD = CompareWSandWD (nws, nwd)
    actorEvalValue = CombineContributions (weightSD, weightWSWD)
}

return actorEvalValue

```

**Figure II.14 – Example: Qualitative CalculateActorEvaluation algorithm**

**Table II.4 – WeightedImportance function for the computation of one element value**

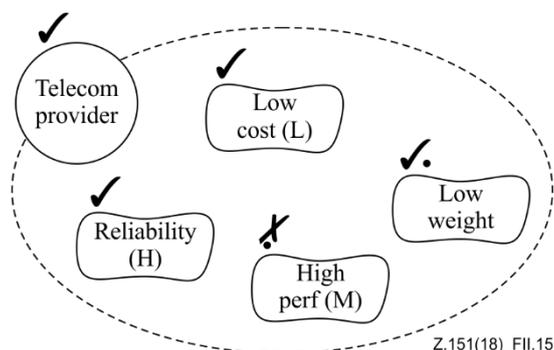
	<i>Denied</i>	<i>WeaklyDenied</i>	<i>WeaklySatisfied</i>	<i>Satisfied</i>	<i>Conflict</i>	<i>Unknown</i>	<i>None</i>
<i>High</i>	Denied	WeaklyDenied	WeaklySatisfied	Satisfied	Conflict	Unknown	None
<i>Medium</i>	WeaklyDenied	WeaklyDenied	WeaklySatisfied	WeaklySatisfied	Conflict	Unknown	None
<i>Low</i>	WeaklyDenied	None	None	WeaklySatisfied	Conflict	Unknown	None
<i>None</i>	None	None	None	None	None	None	None

For example, Figure II.15 shows an actor with four softgoals, three of which with importance other than *None*. The recalculated qualitative evaluation values are:

- Reliability: WeightedImportance (*High, Satisfied*) = *Satisfied*
- Low Cost: WeightedImportance (*Low, Satisfied*) = *WeaklySatisfied*

- High Perf: WeightedImportance (*Medium, WeaklyDenied*) = *WeaklyDenied*
- Low Weight: WeightedImportance (*None, WeaklySatisfied*) = *None*

The comparison of *Satisfied* and *Denied* results in a 1:0 win and therefore *Satisfied*. The comparison of *WeaklySatisfied* and *WeaklyDenied* results in a 1:1 tie and therefore *None*. Finally, the combined contribution of *Satisfied* and *None* results in the actor evaluation of *Satisfied*.



**Figure II.15 – Example: Qualitative evaluation of actors**

Second, the qualitative evaluation value of the overall GRL model is calculated the same way as the qualitative evaluation value of actors except that the qualitative evaluation values and qualitative importance values of actors are used instead of containable elements.

## II.4 Example of hybrid evaluation algorithm

This hybrid GRL algorithm uses *Integer* values for the evaluation, and hence uses the importanceQuantitative attribute of *GRLContainableElements* and the new quantitativeVal attribute of containable elements initialized from the selected *EvaluationStrategy*. However, unlike the quantitative evaluation algorithm seen in clause II.2, the hybrid algorithm uses the qualitative contribution attribute of *Contribution*. A conversion table defines a mapping from qualitative contributions to integer values representing an equivalent quantitative contribution. Once this conversion is done, the rest of the algorithm is similar to the one in clause II.2.

This is an example where quantitative and qualitative values are mixed. In this example, the discrete scale for contributions has 7 levels instead of 201 levels ([−100..100]) as in the quantitative evaluation algorithm. This may improve the usability of models in domains where the weight of contributions cannot easily be determined with precision.

### II.4.1 Calculating hybrid evaluations for decomposition links

This corresponds to the CalculateDecompositions(element) step in Figure II.2. The algorithm is the same as the quantitative algorithm in clause II.2.1.

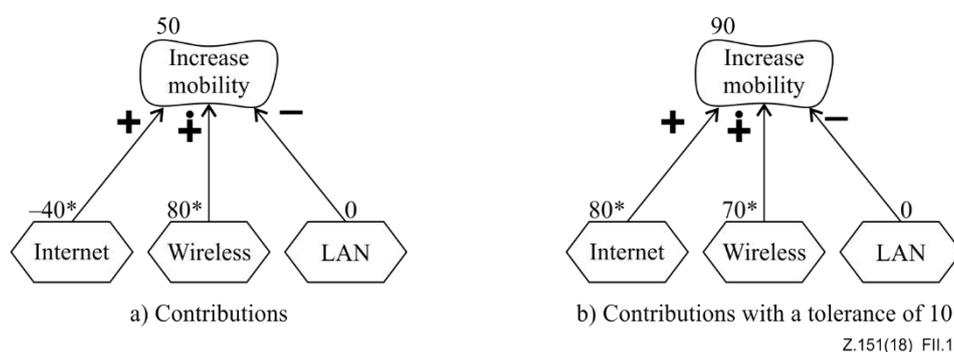
### II.4.2 Calculating hybrid evaluations for contribution links

This corresponds to the CalculateContributions(element, decompValue) step in Figure II.2. The algorithm first maps all qualitative contributions to quantitative contributions using Table II.5. The content of this table reflects the relative ordering of qualitative contributions, however, the associated quantitative numbers could be defined otherwise (e.g., 67 instead of 75, 33 instead of 25, etc.). Once all values are integers, the algorithm seen for the quantitative evaluation in clause II.2.2 is used.

**Table II.5 – Quantitative contribution values for qualitative contributions**

<i>Qualitative contribution</i>	<i>Quantitative contribution</i>
Make	100
<i>SomePositive</i>	75
<i>Help</i>	25
<i>Unknown</i>	0
<i>Hurt</i>	-25
<i>SomeNegative</i>	-75
<i>Break</i>	-100

Figure II.16 provides two examples with three contributions each (the initial decompValue is 0). Strategies initialize two elements in each example. The qualitative contributions are mapped to integer values according to Table II.5. In (a),  $((-40 \times 75) + (80 \times 100) + (0 \times -75))/100 = 50$ . In (b), where the tolerance has been set to 10,  $((80 \times 75) + (70 \times 100) + (0 \times -75))/100 = 130$ . However, as there is no fully satisfied weighted contribution, then  $100 - \text{tolerance} = 90$  is output.



**Figure II.16 – Example: Hybrid evaluation of contribution links**

### II.4.3 Calculating hybrid evaluations for dependency links

This corresponds to the CalculateDependencies(element, contribValue) step in Figure II.2. The algorithm is the same as the quantitative algorithm in clause II.2.3.

### II.4.4 Calculating hybrid evaluations for actors and the overall GRL model

This is the third and last step discussed in clause II.1.2. The algorithm is the same as the quantitative algorithm in clause II.2.4.

### II.5 Calculating with exceeding expectations

The exceeds attribute is only propagated if all source elements exceed expectations in an AND context and if at least one source element exceeds expectations in an IOR or XOR context. An AND context is given for AND-type decomposition links, contribution links and dependency links. Furthermore, the calculation of evaluation values for actors and the overall GRL model also operates in an AND context for the purpose of this calculation. An IOR context is given for IOR-type decomposition links and finally and XOR context is given for XOR-type decomposition links. In addition, a warning is generated if more than one source element exceeds expectations in an XOR context.

## Appendix III

### Examples of UCM path traversal mechanisms

(This appendix does not form an integral part of this Recommendation.)

#### III.1 Introduction

Implementers of a UCM path traversal mechanism may develop their own traversal algorithm, optimizing or extending various aspects of it according to their needs, as long as the resulting traversal mechanism complies with the requirements stated in clause 11.2.2. This appendix shows the traversal of a UCM scenario with the help of the example UCM model introduced in Figure 61. The chosen scenario, defined in clause 8.5.2, is "TL pin TCS success" (a successful call where the PIN for the TL feature is correctly entered and the call is not on the TCS list). Two different traversal mechanisms, a depth-first approach and a breadth-first approach, are illustrated, but the detailed behaviour of the traversal mechanism for each type of path node is beyond the scope of this appendix.

Each path node in the UCM model keeps track of how often it is visited (*visitNumber*) during the traversal of the current scenario and when it was visited with the help of a list of sequence numbers (*sequenceNumbers*). Initially, *visitNumber* is set to zero for all path nodes, *sequenceNumbers* is set to empty, and the value of all global variables is set to "undefined". Furthermore, the included and including scenarios are merged into a new, bigger scenario, and the variables are initialized according to the new scenario. The new scenario is the input to the actual traversal mechanism. It makes reference to the UCM model, which is hence provided as input indirectly. Finally, to safe-guard against infinite loops, the maximum number of visits to a path node is limited to *maxVisits*.

The results of the traversal mechanism are the values of the *visitNumber* and *sequenceNumbers* attributes of the path nodes in the UCM model and any warnings or errors that may have been issued by the traversal.

#### III.2 Example of depth-first UCM path traversal mechanism

A detailed description of a depth-first UCM path traversal mechanism is shown in Figure III.1. First, the start points of the scenario are assigned to the list of currently visited path nodes. Second, the traversal mechanism checks the preconditions of the scenario. Then, the traversal mechanism attempts to move to the next path nodes from the first element in the list of currently visited path nodes. The first element is removed from the list of currently visited path nodes.

If it is not possible to move to other path nodes from the first element, the first element is added to the list of blocked path nodes. The traversal mechanism then attempts to move to the next path nodes from the next element in the list of currently visited path nodes, and so on.

If it is possible to move to other path nodes, then these path nodes are added at the beginning of the list of enabled path nodes. The first path node in the list of enabled path nodes is then removed and added at the beginning of the list of currently visited path nodes. That is, the traversal mechanism is moving from the current path node to the next path node (or one of the next path nodes, if more than one are enabled from the current path node). Adding path nodes at the beginning of the list of enabled path nodes and at the beginning of the list of currently visited path nodes ensures a depth-first approach.

This continues until either an exception is thrown or all enabled and current path nodes are exhausted. At the end of the traversal, the traversal mechanism checks whether the traversal did not get blocked somewhere, whether all postconditions are fulfilled and whether all end points of the scenario were reached.

**Algorithm** UCMPATHTraversalMechanism-DepthFirst**Input** scenario:ScenarioDef // the merged scenario**Output** warningsAndErrors:List // warnings and errors issued during the traversal

```
List currentPathNodes = scenario.startPoints // currently visited path nodes
List enabledPathNodes = ∅ // path nodes that can be visited next
List blockedPathNodes = ∅ // path nodes that cannot continue
warningsAndErrors = ∅ // initially empty
List enabledPathNodesFromCurrent = ∅ // temporary variable
PathNode current, nextCurrent // temporary variables
Integer sequenceNumber = 1 // sequence variable keeps track of the
// number of traversed path nodes
```

**for each** startPoint:PathNode **in** currentPathNodes

startPoint.visitNumber++

try

{

// if preconditions evaluate to false, stop traversal

**if** (!EvaluatePreconditions(scenario))**throw new** TraversalException("warning: preconditions not satisfied")**while** (currentPathNodes.size() > 0)

{

// remove the first path node from the list of currently visited path nodes

current = currentPathNodes.removeFirstElement()

// find the enabled path nodes from the current path node based on

// the path continuation criteria (PCC)

enabledPathNodesFromCurrent = GetEnabledPathNodes(current)

**if** (enabledPathNodesFromCurrent.size() == 0)

{

// the current path node is blocked → continue with another current

// path node in while loop

blockedPathNodes.addAtEnd(current)

// add sequence number to an end point of the scenario; to avoid

// duplicate sequence numbers for end points, this is only done if the

// number of sequence numbers does not exceed the number of visits

**if** ((current **is of type** EndPoint) **and**

(current.visitNumber &gt; current.sequenceNumbers.size()))

{

current.sequenceNumbers.add(sequenceNumber);

sequenceNumber++;

}

}

else

{

// add found path nodes to the list of enabled path nodes

enabledPathNodes.addAtBeginning(enabledPathNodesFromCurrent)

// add sequence number to path node when the traversal is ready to

// move on to the next path nodes



**return** warningsAndErrors

**Figure III.1 – Example: Depth-first UCM path traversal mechanism**

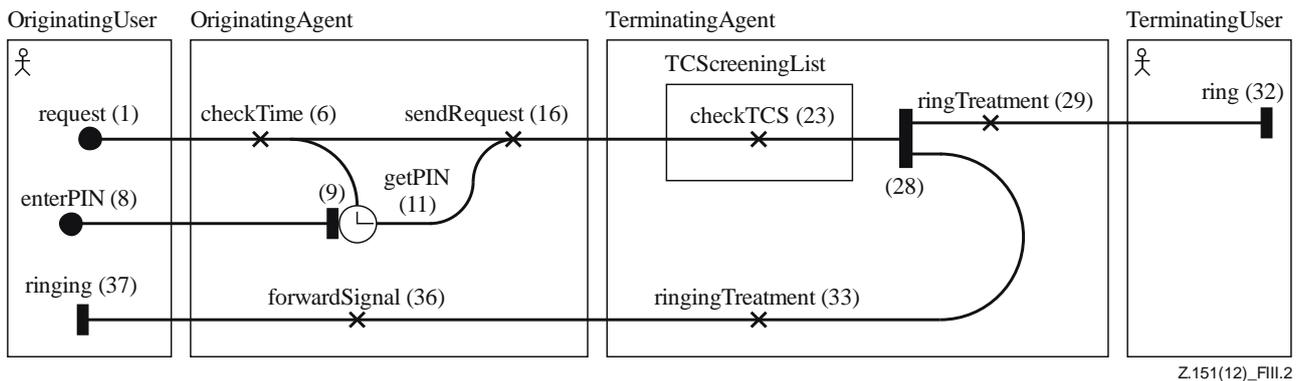
Given the depth-first algorithm from Figure III.1, the "TL pin TCS success" scenario from the example in clause 8.5.2 is traversed as follows (variables are only shown if they changed; sequence numbers are shown in parentheses):

- currentPathNodes = {request, enterPIN}, enabledPathNodes =  $\emptyset$ , blockedPathNodes =  $\emptyset$ , and preconditions of the scenario evaluate to true as none are specified.
- Move from request start point (1) to Originating stub: currentPathNodes = {Originating, enterPIN}.
- Move from Originating stub (2) to start point of the Originating feature plug-in map: currentPathNodes = {start, enterPIN}.
- Move from start point of the Originating feature plug-in map (3) to OrigFeatures stub: currentPathNodes = {OrigFeatures, enterPIN}.
- Move from OrigFeatures stub (4) to start point of Teen Line plug-in map because this is the only plug-in map with a precondition evaluating to true (subTL): currentPathNodes = {start, enterPIN}.
- Move from start point (5) of Teen Line plug-in map to checkTime responsibility: currentPathNodes = {checkTime, enterPIN}.
- Move from checkTime responsibility (6) to "TLactive?" OR-fork: currentPathNodes = {"TLactive?" OR-fork, enterPIN}.
- Move from "TLactive" OR-fork (7) to getPIN timer because the condition of this branch (TLactive) evaluates to true: currentPathNodes = {getPIN timer, enterPIN}.
- getPIN timer is blocked: currentPathNodes = {enterPIN}, blockedPathNodes = {getPIN timer}.
- Move from enterPIN start point (8) to getPIN end point: currentPathNodes = {getPIN end point, getPIN timer}, blockedPathNodes =  $\emptyset$  (because all blockedPathNodes are moved to currentPathNodes if a new path node is added to the currentPathNodes).
- Move from getPIN end point (9) to getPIN connect: currentPathNodes = {getPIN connect, getPIN timer}.
- Move from getPIN connect (10) to getPIN timer: currentPathNodes = {getPIN timer} (because the second timer path node in currentPathNodes was removed as a timer is a synchronization point of two paths).
- Move from getPIN timer (11) to "PINvalid?" OR-fork because the trigger path arrived: currentPathNodes = {"PINvalid?" OR-fork}.
- Move from "PINvalid?" OR-fork (12) to OR-join because the condition of this branch (PINvalid) evaluates to true: currentPathNodes = {OR-join}.
- Move from OR-join (13) to success end point: currentPathNodes = {success}.
- Move from success end point (14) to OUT1 out-path of OrigFeatures stub: currentPathNodes = {OrigFeatures}.
- Move from OUT1 out-path of OrigFeatures stub (15) to sendRequest responsibility: currentPathNodes = {sendRequest}.
- Move from sendRequest responsibility (16) to success end point: currentPathNodes = {success}.

- Move from success end point (17) to OUT1 out-path of Originating stub: `currentPathNodes = {Originating}`.
- Move from OUT1 out-path of Originating stub (18) to Terminating stub: `currentPathNodes = {Terminating}`.
- Move from Terminating stub (19) to start point of Terminating Features plug-in map: `currentPathNodes = {start}`.
- Move from start point of Terminating Features plug-in map (20) to TermFeatures stub: `currentPathNodes = {TermFeatures}`.
- Move from TermFeatures stub (21) to start point of Terminating Call Screening plug-in map because this is the only plug-in map with a precondition evaluating to true (subTCS): `currentPathNodes = {start}`.
- Move from start point of Terminating Call Screening plug-in map (22) to checkTCS responsibility: `currentPathNodes = {checkTCS}`.
- Move from checkTCS responsibility (23) to "OnTCSList?" OR-fork: `currentPathNodes = {"OnTCSList?" OR-fork}`.
- Move from "OnTCSList?" OR-fork (24) to success end point because the condition of this branch (NotOnTCSList) evaluates to true: `currentPathNodes = {success}`.
- Move from success end point (25) to OUT1 out-path of TermFeatures stub: `currentPathNodes = {TermFeatures}`.
- Move from OUT1 out-path of TermFeatures stub (26) to "Busy?" OR-fork: `currentPathNodes = {"Busy?" OR-fork}`.
- Move from "Busy?" OR-fork (27) to AND-fork because the condition of this branch (NotBusy) evaluates to true: `currentPathNodes = {AND-fork}`.
- Move from AND-fork (28) to ringTreatment responsibility, therefore exploring the first branch of the AND-fork in a depth-first way: `currentPathNodes = {ringTreatment}`, `enabledPathNodes = {ringingTreatment}`.
- Move from ringTreatment responsibility (29) to success end point: `currentPathNodes = {success}`.
- Move from success end point (30) to OUT1 out-path of Terminating stub: `currentPathNodes = {Terminating}`.
- Move from OUT1 out-path of Terminating stub (31) to ring end point: `currentPathNodes = {ring}`.
- ring end point (32) is blocked because an end point of the scenario has been reached; therefore, explore the second branch of the AND-fork by moving from AND-fork to ringingTreatment responsibility: `currentPathNodes = {ringingTreatment, ring}`, `enabledPathNodes = ∅` (because the next enabledPathNode is taken from the list and moved to currentPathNodes; this also causes the blockedPathNode ring to be added back to the currentPathNodes).
- Move from ringingTreatment responsibility (33) to reportSuccess end point: `currentPathNodes = {reportSuccess, ring}`.
- Move from reportSuccess end point (34) to OUT2 out-path of Terminating stub: `currentPathNodes = {Terminating stub, ring}`.
- Move from OUT2 out-path of Terminating stub (35) to forwardSignal responsibility: `currentPathNodes = {forwardSignal, ring}`.
- Move from forwardSignal responsibility (36) to ringing end point: `currentPathNodes = {ringing, ring}`.

- ringing end point (37) is blocked because an end point of the scenario has been reached:  $\text{currentPathNodes} = \{\text{ring}\}$ ,  $\text{blockedPathNodes} = \{\text{ringing}\}$ .
- try next path node in  $\text{currentPathNodes}$ ; ring end point is blocked because an end point of the scenario has been reached:  $\text{currentPathNodes} = \emptyset$ ,  $\text{blockedPathNodes} = \{\text{ringing}, \text{ring}\}$ .
- while loop terminates.
- no warnings and errors are issued because a) only end points remain in  $\text{currentPathNodes} = \emptyset$  and  $\text{blockedPathNodes} = \{\text{ringing}, \text{ring}\}$ , b) the end points of the scenario (ringing, ring) were visited, and c) all postconditions are fulfilled.

The flattened UCM model that results from the traversal of the "TL pin TCS success" scenario is shown in Figure III.2. The sequence number assigned by the traversal mechanism is indicated for each path node in parentheses. The gaps in the sequence numbers are explained by the fact that the flattened UCM model does not show all encountered path elements, i.e., path elements that are either a stub, a start point bound to an in-path of a stub by a plug-in binding or an end point bound to an out-path of a stub by a plug-in binding. Furthermore, the flattened UCM model shows resolved component plug-in bindings.



Z.151(12)\_FIII.2

**Figure III.2 – Example: Flattened UCM model of depth-first traversed scenario**

### III.3 Example of breadth-first UCM path traversal mechanism

The difference between the depth-first approach shown in Figure III.1 and the breadth-first approach shown in Figure III.3 is that in one step the traversal mechanism attempts to move to the next path nodes not only from the first currently visited path node but from all currently visited path nodes. Therefore, the algorithm contains a sequence of two *for* loops. The first identifies all enabled path nodes given the set of currently visited path nodes. The second moves the traversal from the current path nodes to the enabled path nodes all at once. Adding path nodes at the end of the list of enabled path nodes and at the end of the list of currently visited path nodes ensures a breadth-first approach. The treatment of the start points of the scenario is also slightly different compared to the depth-first approach as the start points are added only one by one to the list of currently visited path nodes.

```

Algorithm UCMPATHTraversalMechanism-BreadthFirst
Input scenario:ScenarioDef // the merged scenario
Output warningsAndErrors:List // warnings and errors issued during the traversal

List currentPathNodes = ∅ // currently visited path nodes
List enabledPathNodes = ∅ // path nodes that can be visited next
List blockedPathNodes = ∅ // path nodes that cannot continue
warningAndErrors = ∅ // initially empty
List enabledPathNodesFromCurrent = ∅ // temporary variable

```

```

PathNode current, nextCurrent           // temporary variables
Integer sequenceNumber = 1              // sequence variable keeps track of the
                                        // number of traversed path nodes

try
{
    // if preconditions evaluate to false, stop traversal
    if (!EvaluatePreconditions(scenario))
        throw new TraversalException("warning: preconditions not satisfied")

    // add the first start point to the list of currently visited path nodes if it exists
    current = scenario.getNextStartPoint()
    if (current != null)
    {
        current.visitNumber++
        currentPathNodes.add(current)
    }
    while (currentPathNodes.size() > 0)
    {
        // go through all current path nodes at once to discover enabled path nodes
        for each pn:PathNode in currentPathNodes
        {
            // find the enabled path nodes from the current path node based on
            // the path continuation criteria (PCC)
            enabledPathNodesFromCurrent = GetEnabledPathNodes(pn)
            if (enabledPathNodesFromCurrent.size() == 0)
            {
                // the current path node is blocked → continue with another
                // current path node in for loop
                blockedPathNodes.addAtEnd(pn)
                // add sequence number to an end point of the scenario; to
                // avoid duplicate sequence numbers for end points, this is
                // only done if the number of sequence numbers does not
                // exceed the number of visits
                if ((pn is of type EndPoint) and
                    (pn.visitNumber > pn.sequenceNumbers.size()))
                {
                    pn.sequenceNumbers.add(sequenceNumber);
                    sequenceNumber++;
                }
            }
            else
            {
                // add found path nodes to the list of enabled path nodes
                enabledPathNodes.addAtEnd(enabledPathNodesFromCurrent)
                // add sequence number to path node when the traversal is
                // ready to move on to the next path nodes
                pn.sequenceNumbers.add(sequenceNumber);
                sequenceNumber++;
            }
        }
        currentPathNodes.clear()
        // at once move all enabled path nodes to the current path nodes
    }
}

```

```

for each pn:PathNode in enabledPathNodes
{
    // not shown here but this is the place where any additional
    // behaviour for the pn path node is executed depending on the type
    // of path node; for more information on what may occur here see
    // the depth-first algorithm

    // add the pn path node at the end of the list of currently visited path
    // nodes, so that the traversal continues in a breadth-first way
    currentPathNodes.addAtEnd(pn)
    pn.visitNumber++
    if (pn.visitNumber > maxVisit)
        throw new TraversalException("warning: infinite loop")
}
enabledPathNodes.clear();
if (currentPathNodes.size() > 0)
{
    // since moving to new path nodes may impact already blocked
    // path nodes, the blocked path nodes are added back to the
    // currently visited path nodes; they are added at the end of the list
    // to ensure a breadth-first approach
    currentPathNodes.addAllAtEnd(blockedPathNodes)
    blockedPathNodes.clear()
}
else
{
    // the traversal did not move to a new path node, therefore try the
    // next start point of the scenario
    current = scenario.getNextStartPoint()
    if (current != null)
    {
        current.visitNumber++
        currentPathNodes.add(current)
    }
}
}
}
catch (TraversalException te)
{
    warningsAndErrors.add(te.getMessage())
}

// verify successful completion of traversal
for each pn:PathNode in (currentPathNodes  $\cup$  blockedPathNodes)
    if (pn is not of type EndPoint)
        warningsAndErrors.add("warning: traversal not at an end point")
    else if (!EvaluateCondition(pn.postcondition))
        warningsAndErrors.add("error: postcondition of end point is false")
for each endPoint:PathNode in scenario.endPoints
    if (endPoint.visitNumber == 0)
        warningsAndErrors.add("error: end point not reached")
for each postcondition:Condition in scenario.postconditions ()

```

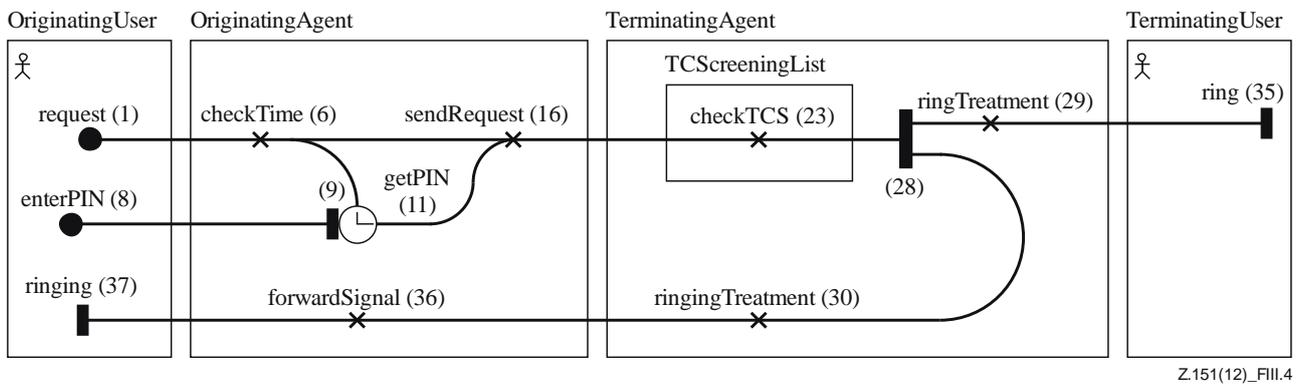
```
    if (!EvaluateCondition(postcondition))
        warningsAndErrors.add("error: postcondition is false")
return warningsAndErrors
```

**Figure III.3 – Example: Breadth-first UCM path traversal mechanism**

Given the breadth-first algorithm from Figure III.3, the "TL pin TCS success" scenario from the example in clause 8.5.2 is traversed as follows (only the parts that changed from the depth-first approach are shown; sequence numbers are shown in parentheses):

- Preconditions of the scenario evaluate to true as none are specified, `currentPathNodes = {request}`, `enabledPathNodes = ∅`, and `blockedPathNodes = ∅`.
- The same as the depth-first approach except `enterPIN` is not on the `currentPathNodes` list up to where `getPIN` timer is blocked: `currentPathNodes = {enterPIN}`, `blockedPathNodes = {getPIN timer}` (because the next start point of the scenario is added to `currentPathNodes`).
- The same as the depth-first approach until the AND-fork is reached; move from AND-fork (28) to `ringTreatment` responsibility and `ringingTreatment` responsibility, therefore exploring both branches of the AND-fork in a breadth-first way: `currentPathNodes = {ringTreatment, ringingTreatment}`.
- Move from `ringTreatment` responsibility (29) to success end point and `ringingTreatment` responsibility (30) to `reportSuccess` end point: `currentPathNodes = {success, reportSuccess}`.
- Move from success end point (31) to OUT1 out-path of Terminating stub and `reportSuccess` end point (32) to OUT2 out-path of Terminating stub: `currentPathNodes = {Terminating, Terminating}`.
- Move from OUT1 out-path of Terminating stub (33) to ring end point and OUT2 out-path of Terminating stub (34) to `forwardSignal` responsibility: `currentPathNodes = {ring, forwardSignal}`.
- ring end point (35) is blocked because an end point of the scenario has been reached; move from `forwardSignal` responsibility (36) to ringing end point: `currentPathNodes = {ringing, ring}` (because all `blockedPathNodes` are moved to `currentPathNodes` if a new path node is added to the `currentPathNodes`).
- ringing end point (37) is blocked because an end point of the scenario has been reached; ring is blocked because an end point of the scenario has been reached: `currentPathNodes = ∅`, `blockedPathNodes = {ringing, ring}`.
- while loop terminates.
- no warnings and errors are issued because a) only end points remain in `currentPathNodes = ∅` and `blockedPathNodes = {ringing, ring}`, b) the end points of the scenario (`ringing, ring`) were visited, and c) all postconditions are fulfilled.

The flattened UCM model that results from the breadth-first traversal of the "TL pin TCS success" scenario is the same as the one for the depth-first traversal, as shown in Figure III.2, except for the sequence numbers of `ring` and `ringingTreatment`. The difference can be seen in Figure III.4 for the path nodes that follow the AND-fork.



**Figure III.4 – Example: Flattened UCM model of breadth-first traversed scenario**

## URN Change Request Form

Please fill in the following details		
Character of change:	<input type="checkbox"/> error correction	<input type="checkbox"/> clarification
	<input type="checkbox"/> simplification	<input type="checkbox"/> extension
	<input type="checkbox"/> modification	<input type="checkbox"/> decommission
Short summary of the change request.		
Short justification of the change request.		
Have you consulted other users?	<input type="checkbox"/> yes	<input type="checkbox"/> no
Is this view shared in your organization?	<input type="checkbox"/> yes	<input type="checkbox"/> no
How many users do you represent?	<input type="checkbox"/> 1-5	<input type="checkbox"/> 6-10
	<input type="checkbox"/> 11-100	<input type="checkbox"/> over 100
Your name and address		

*Please attach further sheets with details if necessary*

URN (ITU-T Z.151) Rapporteur, c/o ITU-T, Place des Nations, CH-1211, Geneva 20, Switzerland.  
Fax: +41 22 730 5853, e-mail: [urn.rapporteur@ties.itu.int](mailto:urn.rapporteur@ties.itu.int).

## **Bibliography**

Many papers and theses related to URN, GRL and UCM are available online at the following location:

URN virtual library, <<http://www.UseCaseMaps.org/pub/>>.



## SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	Tariff and accounting principles and international telecommunication/ICT economic and policy issues
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Environment and ICTs, climate change, e-waste, energy efficiency; construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling, and associated measurements and tests
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects, next-generation networks, Internet of Things and smart cities
<b>Series Z</b>	<b>Languages and general software aspects for telecommunication systems</b>