International Telecommunication Union

ITU-T Z.151

(11/2008)

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

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

International Telecommunication Union

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
User Requirements Notation (URN)	**Z.150–Z.159**
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).

Source

Recommendation ITU-T Z.151 was approved on 13 November 2008 by ITU-T Study Group 17 (2009-2012) under Recommendation ITU-T A.8 procedure.

Keywords

Evaluation, functional requirements, goals, graphical notation, non-functional requirements, rationales, scenarios, specification technique.

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/.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

Introduction

A) Coverage: URN has concepts for the specification of goals, non-functional requirements, rationales, 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.

B) 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 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 goals and requirements description to high-level system design and architecture.

C) 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
- Appendix I: Summary of the URN Notation
- 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, 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 goals and requirements description 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 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 three main categories of concepts in GRL: *actors*, *intentional elements*, 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, 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. 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 us 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 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. These 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 may aid in identifying 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 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 UCM specifications may be refined into more detailed scenario models such as MSCs and UML sequence diagrams, or into state machines in SDL 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 modeling 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 (run-time) refinement capabilities of the UCM notation allow for the specification of (run-time) 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 syntax, and semantics of GRL features.
- Clause 8 specifies the abstract syntax, concrete syntax, and semantics of UCM features.
- 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.
- 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.100] Recommendation ITU-T Z.100 (2007), *Specification and Description Language (SDL)*.

[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 (2003), *User Requirements Notation (URN) – Language requirements and framework*.

[W3C XSD1] XML Schema Part 1 (2008): *Structures Second Edition*. <http://www.w3.org/TR/xmlschema-1>

[W3C XSD2] XML Schema Part 2 (2008): *Datatypes Second Edition*. <http://www.w3.org/TR/xmlschema-2>

3 Definitions

The definitions given in [ITU-T Z.150] apply, with the following additions:

3.1 **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 **goal-oriented requirement language**: The subset of the User Requirements Notation used to model and analyse non-functional requirements and quality attributes with goal graphs.

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

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

3.5 **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.

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

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

3.8 **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.

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

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

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

3.12 **unconnected start point**: (UCM) start point that is not directly connected to another end point or to another path.

3.13 **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.

3.14 **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^{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.

3.15 **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
FR	Functional Requirement
GRL	Goal-oriented Requirement Language
MSC	Message Sequence Chart
NFR	Non-Functional Requirement
PCC	Path Continuation Criteria
PT	Path Traversal
SDL	Specification and Description Language
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
XML	eXtensible Markup Language
XSD	XML Schema Definition

5 Conventions

The text of this clause is not normative. Instead, it defines the conventions used for describing the User Requirements Notation.

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 defined 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. For each metaclass in the metamodel, attributes, relationships to other metaclasses, and constraints (static conditions expressed in natural language) are specified.

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 color 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.

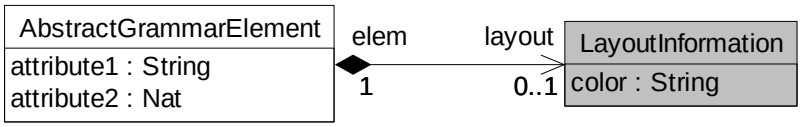


Figure 1 – Example metaclasses from an abstract grammar (white) and a concrete grammar (grey)

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 top-most metaclass, *URNspec* (see Figure 2), contains directly or indirectly all the other elements of a URN model, including concerns (see Figure 3). In this Recommendation, the terms "URN model" and "URN specification" are used interchangeably.

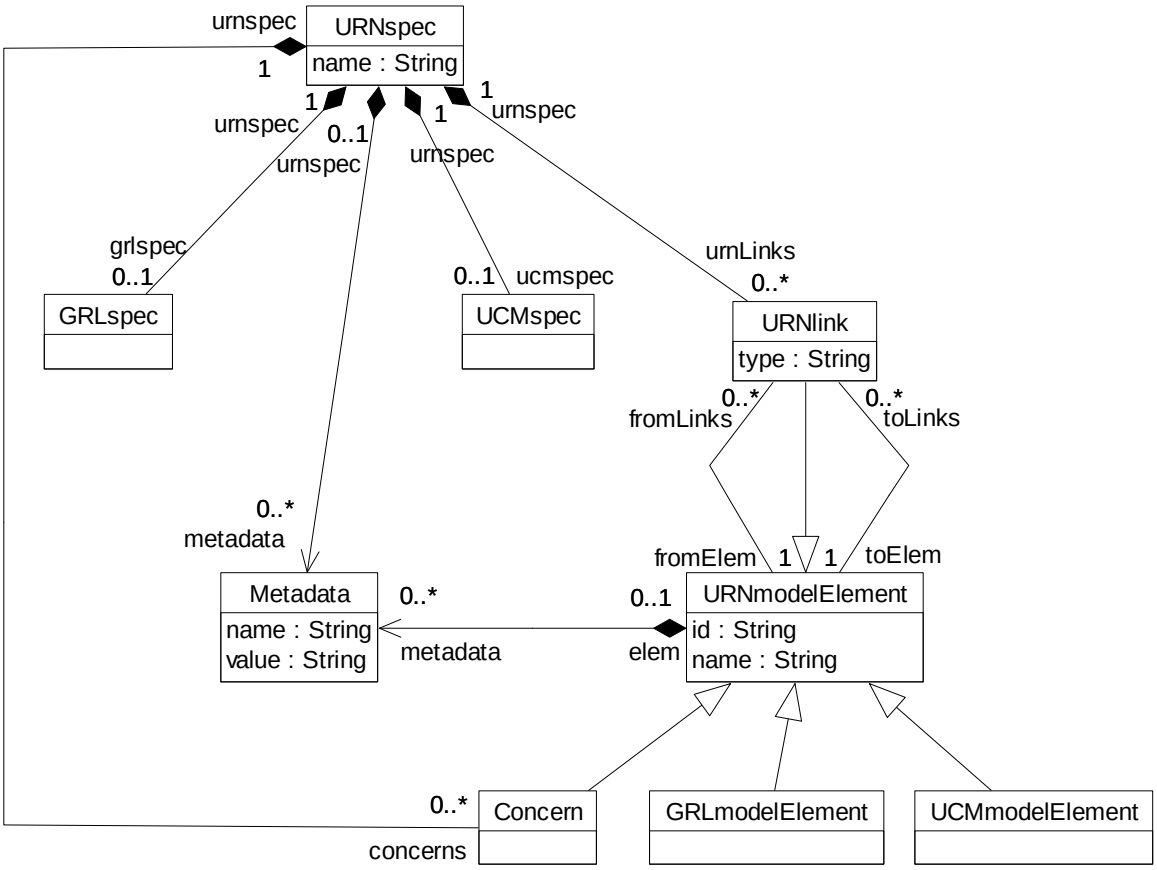


Figure 2 – Abstract grammar: URN specification, links, metadata and model elements

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 2).

a) Abstract grammar

Attributes

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

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.

Constraints

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

b) Concrete grammar

URNspec has no concrete syntax. However, it may contain additional information in an instance of *ConcreteURNspec*.

Relationships

- Composition of *ConcreteURNspec* (0..1): The *URNspec* may contain one concrete URN specification (see Figure 4).

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 2).

a) Abstract grammar

Attributes

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

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).

Constraints

a. id must be unique within the URN specification.

b. All instances of *URNmodelElement* must appear in one of its subclasses (that is, metaclass *URNmodelElement* is abstract).

b) Concrete grammar

The concrete syntax for *URNmodelElement* is further defined in its subclasses. In addition, a *URNmodelElement* may contain an informal description in a *Description*.

Relationships

- Composition of *Description* (0..1): A *URNmodelElement* may contain one description (see Figure 5).

c) Semantics

A *URNmodelElement* is a uniquely identifiable model element that can contain metadata and be linked to other model elements. Its subclasses may 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 2).

a) Abstract grammar

Attributes

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

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.

Constraints

a. 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.

Relationships

- Inherits relationships from *URNmodelElement*.

c) Semantics

*URNlink*s 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 2).

a) Abstract grammar

Attributes

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

Relationships

None.

Constraints

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

b) Concrete grammar

None.

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 Figure 2 and Figure 3).

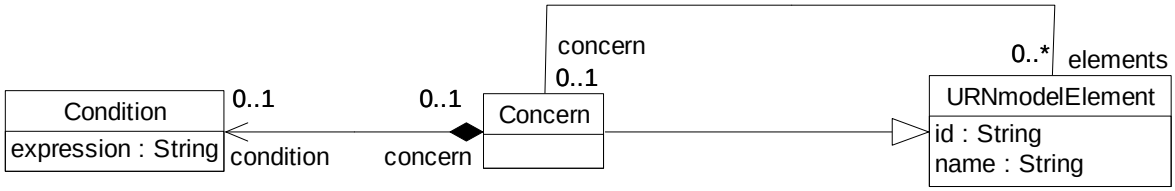


Figure 3 – Abstract grammar: URN concerns

a) Abstract grammar

Attributes

- Inherits attributes from *URNmodelElement*.

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* may group URN model elements.

Constraints

a. Inherits constraints from *URNmodelElement*.

b) Concrete grammar

A *Concern* does not have a visual representation.

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, or postcondition. A *Condition* is contained by one of the following model elements: *Concern*, *StartPoint*, *EndPoint*, *PluginBinding*, *NodeConnection*, or *ScenarioDef* (see Figure 3, Figure 68, and Figure 83).

a) Abstract grammar

Attributes

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

Relationships

None.

Constraints

a. The expression of a *Condition* must be a Boolean expression, as defined in clause 9.3.

b. 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 *NodeConnection*s, *StartPoint*s, and *EndPoint*s (See clauses 8.2.3, 8.2.6, and 8.2.7, respectively). Conditions for *Concern*s, *PluginBinding*s, or *ScenarioDef*s are not visualized.

Relationships

- Composition of *ConcreteCondition* (0..1): A *Condition* may have one concrete condition (see Figure 6).
- Composition of *Label* (0..1): A *Condition* may have one label (see Figure 46).

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 *Variable*s and must 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 post-conditions 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.

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 4).

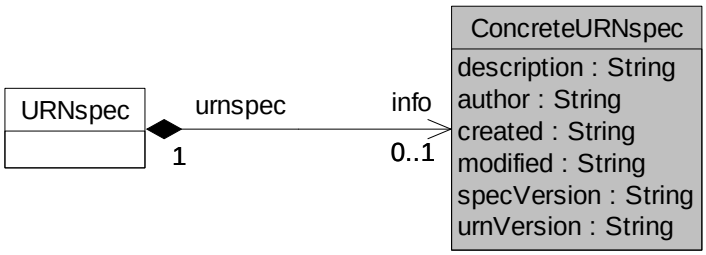


Figure 4 – Concrete grammar: ConcreteURNspec metaclass

a) Abstract grammar

None. This is a concrete syntax metaclass only.

b) Concrete grammar

There is no visual representation of this metaclass.

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)".

Constraints

a. The date modified is later than the date created.

6.2.2 Description

An informal *Description* can be attached to any *URNmodelElement* (see Figure 5).

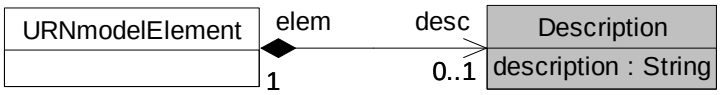


Figure 5 – Concrete grammar: Description metaclass

a) Abstract grammar

None. This is a concrete syntax metaclass only.

b) Concrete grammar

There is no visual representation of this metaclass.

Attributes

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

Constraints

None.

6.2.3 ConcreteCondition

ConcreteCondition defines a label and a description for a *Condition* (see Figure 6).

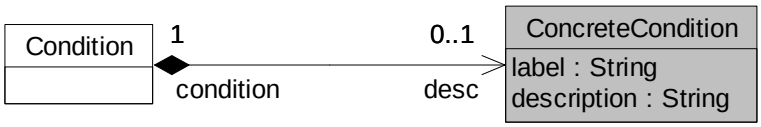


Figure 6 – Concrete grammar: ConcreteCondition metaclass

a) Abstract grammar

None. This is a concrete syntax metaclass only.

b) Concrete grammar

The label of *ConcreteCondition* is visualized for containing *Condition*s of *NodeConnection*s, *StartPoint*s, and *EndPoint*s (see clauses 8.2.3, 8.2.6, and 8.2.7, respectively).

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 concrete grammar metaclasses: clause 7.6.

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 7.6.

7.1.1 GRLspec

GRLspec serves as a container for the GRL specification elements (see Figure 7).

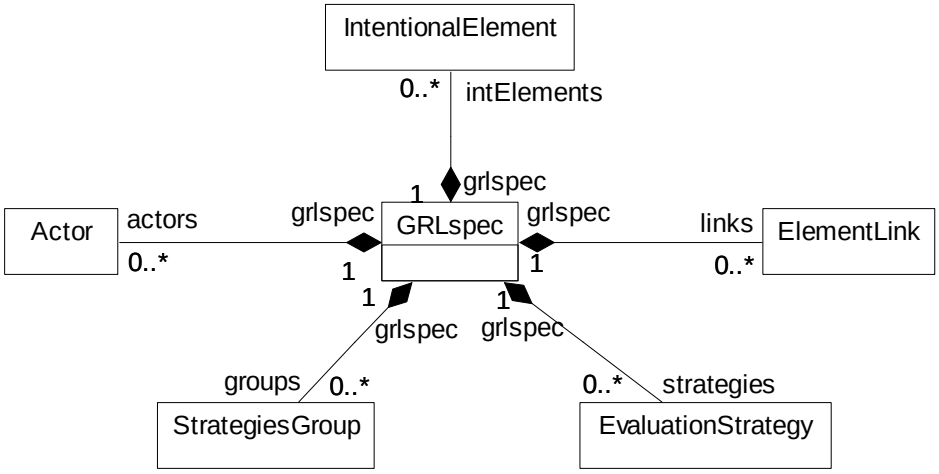


Figure 7 – Abstract grammar: GRL specification

a) Abstract grammar

Attributes

None.

Relationships

- Contained by *URNspec* (1): The *GRLspec* is contained in the URN specification (see Figure 2).
- Composition of *Actor* (0..*): The *GRLspec* may contain actor definitions.
- Composition of *IntentionalElement* (0..*): The *GRLspec* may contain intentional elements.
- 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.

Constraints

None.

b) Concrete grammar

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

Relationships

- Composition of *ConcreteGRLspec* (0..1): The *GRLspec* may contain one concrete GRL specification (see Figure 25).
- Composition of *GRLGraph* (0..*): The *GRLspec* may contain GRL graphs (see Figure 25).

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 8).

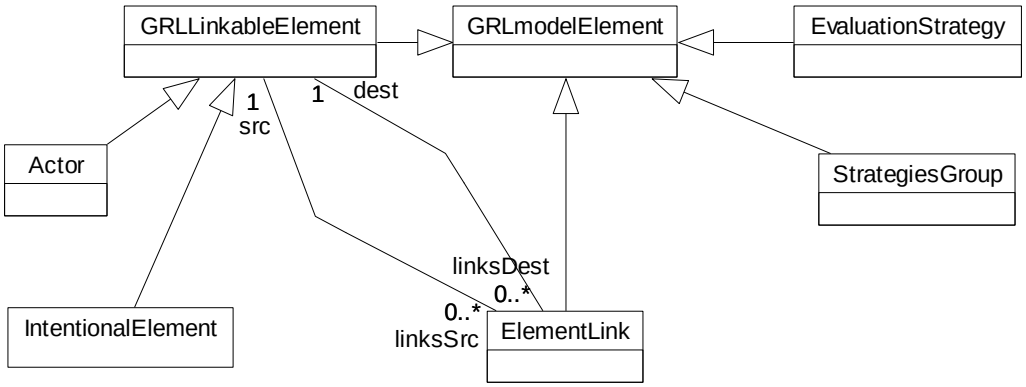


Figure 8 – Abstract grammar: GRL model elements and linkable elements

Attributes

- Inherits attributes from *URNmodelElement* (see Figure 2).

Relationships

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

Constraints

a. Inherits constraints from *URNmodelElement*.

b. All instances of *GRLmodelElement* must 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.

Relationships

- Inherits relationships from *URNmodelElement* (see Figure 2).
- *GRLmodelElement* is a superclass of *GRLGraph*, *ActorRef*, *GRLNode*, and *LinkRef*.

c) Semantics

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

7.1.3 GRLLinkableElement

A *GRLLinkableElement* is a GRL model element that can be linked to other GRL linkable elements through an *ElementLink*. *GRLLinkableElement* abstracts the commonalities of actor definitions and intentional elements (see Figure 8).

a) Abstract grammar

Attributes

- Inherits attributes from *GRLmodelElement*.

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 *IntentionalElement* and *Actor*.

Constraints

a. Inherits constraints from *GRLmodelElement*.

b. All instances of *GRLLinkableElement* must appear in one of its subclasses (that is, metaclass *GRLLinkableElement* is abstract).

b) Concrete grammar

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

Relationships

- Inherits relationships from *GRLmodelElement*.

c) Semantics

A *GRLLinkableElement* is a GRL model element that can be linked to other actor definitions and intentional elements.

7.2 GRL actors

Figure 9 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.

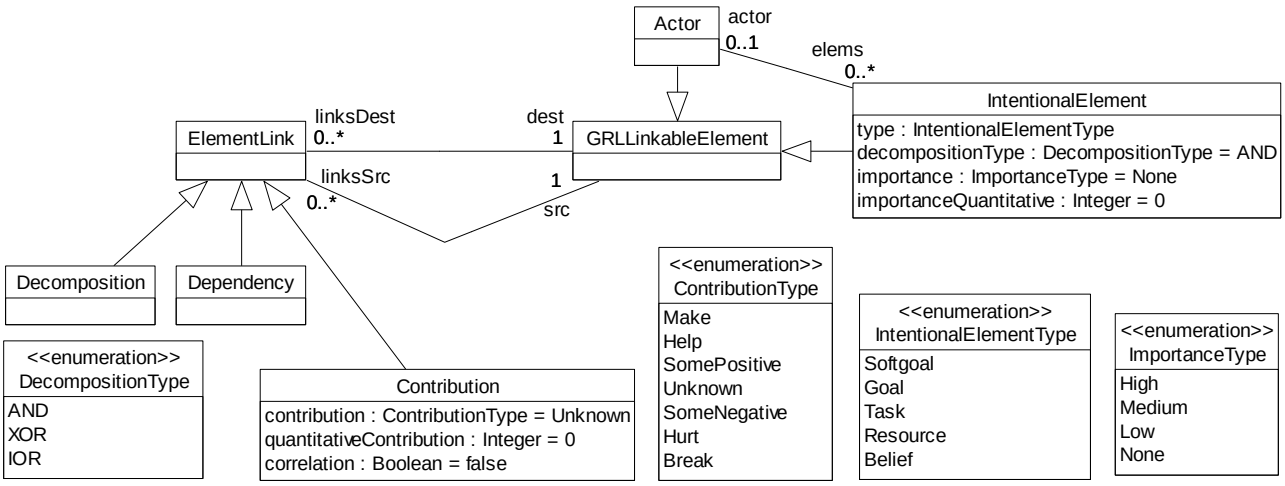


Figure 9 – Abstract grammar: GRL actors, intentional elements, and links

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 intentional elements (see Figure 9).

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

a) Abstract grammar

Attributes

- Inherits attributes from *GRLLinkableElement*.

Relationships

- Inherits relationships from *GRLLinkableElement*.
- Contained by *GRLspec* (1): An *Actor* definition is contained in the GRL specification (see Figure 7).
- Association with *IntentionalElement* (0..*): An *Actor* definition may contain intentional elements.

Constraints

a. Inherits constraints from *GRLLinkableElement*.

b. Any two *Actor* definitions cannot share the same name inside a URN specification.

c. The name of an *Actor* definition cannot be an empty String.

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 color of an actor definition's circle line and the fill color are defined in the actor definition's concrete style (*ConcreteStyle*) and are hence shared by all the actor's references.

Relationships

- Inherits relationships from *GRLLinkableElement*.
- Composition of *ConcreteStyle* (0..1): An *Actor* definition may contain one concrete style (see Figure 25).
- Association with *ActorRef* (0..*): An *Actor* definition may be referenced by actor references (see Figure 25).
- Association with *CollapsedActorRef* (0..*): An *Actor* definition may be referenced by collapsed actor references (see Figure 25).

c) Semantics

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

d) Model

None.

e) Examples

Figure 10 is a GRL diagram that shows a "Telecom Provider" as a collapsed actor (left) and as an actor with boundary (right). See clauses 7.6.3 and 7.6.6 for the details of the concrete syntax.

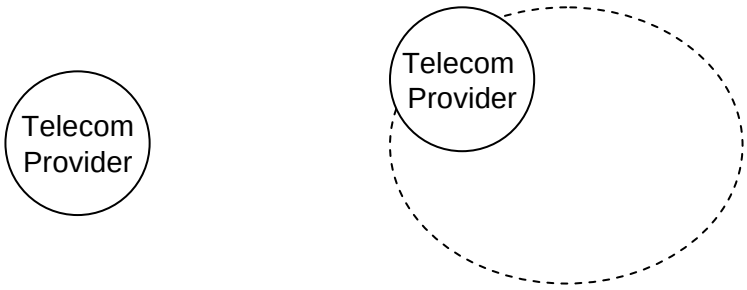


Figure 10 – Example: GRL collapsed actor (left) and actor with boundary (right)

7.3 GRL intentional elements

7.3.1 IntentionalElement

An *IntentionalElement* is a GRL linkable 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 the other. 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. An *IntentionalElement* may be given *Evaluation* values (see Figure 9 and Figure 23).

a) Abstract grammar

Attributes

- Inherits attributes from *GRLLinkableElement*.
- 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.
- importance (*ImportanceType*): Qualitative importance of the intentional element to its containing actor definition, if any. Default value is None.
- importanceQuantitative (*Integer*): Quantitative importance of the intentional element to its containing actor definition, if any. Default value is 0 (see clause 9.2.2).

Relationships

- Inherits relationships from *GRLLinkableElement*.
- Contained by *GRLspec* (1): An *IntentionalElement* is contained in the GRL specification (see Figure 7).
- Association with *Actor* (0..1): An *IntentionalElement* may be contained in one actor definition.
- Uses *IntentionalElementType* enumeration.
- Uses *DecompositionType* enumeration.
- Uses *ImportanceType* enumeration.

Constraints

a. Inherits constraints from *GRLLinkableElement*.

b. The name of an *IntentionalElement* cannot be an empty String.

c. importanceQuantitative ≥ 0 and importanceQuantitative ≤ 100.

d. If an *IntentionalElement* is associated with an *Actor* definition, then there is only one *IntentionalElement* with this name associated to the *Actor* definition.

e. 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. The line and fill colors of an intentional element are specified in its definition's concrete style (*ConcreteStyle*) and are hence shared by all the intentional element's references.

Relationships

- Inherits relationships from *GRLLinkableElement*.
- Composition of *ConcreteStyle* (0..1): An *IntentionalElement* definition may contain one concrete style (see Figure 25).
- Association with *IntentionalElementRef* (0..*): An *IntentionalElement* definition may be referenced by intentional element references (see Figure 25).

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.

A value of 0 for importanceQuantitative means that the intentional element 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 is not important to the actor definition. Often, only top-level intentional elements 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 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.

d) Model

None.

e) Examples

Figure 11 is a GRL diagram that shows five intentional elements, one for each type:

Figure 11 – Example: GRL intentional elements

- "Voice Connection Be Setup" is defined as a (hard) goal because this is something than 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 7.3.2, 7.3.3 and 7.6.5 for the details of the concrete syntax.

The GRL diagram in Figure 12 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):

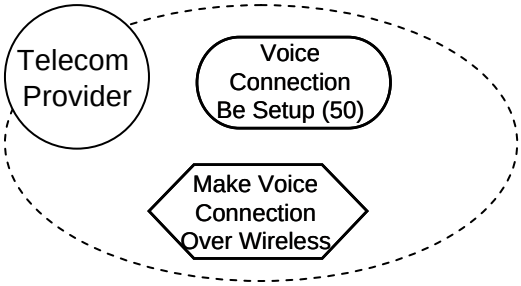


Figure 12 – Example: GRL actor that contains a goal and a task

7.3.2 IntentionalElementType

An intentional element can be a Goal, Softgoal, Task, Resource, or Belief (see Figure 9).

a) Abstract grammar

Attributes

- None (enumeration metaclass).

Relationships

- Used by *IntentionalElement*.

Constraints

None.

b) Concrete grammar

The symbols in Figure 13 are used to denote the various types of GRL intentional elements. See usage in *IntentionalElementRef*, clause 7.6.5.

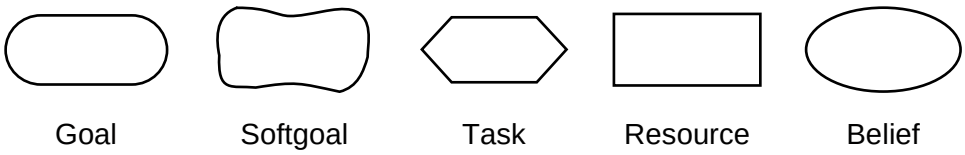


Figure 13 – Symbols: GRL intentional element types

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.

7.3.3 ImportanceType

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

a) Abstract grammar

Attributes

- None (enumeration metaclass).

Relationships

- Used by *IntentionalElement*.

Constraints

None.

b) Concrete grammar

None (enumeration metaclass). However, it influences the presentation of intentional elements contained by actor definitions (see clause 7.6.5).

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 with a None importance will have no impact on the qualitative evaluation of the global satisfaction of the associated actor definition.

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 9).

a) Abstract grammar

Attributes

- Inherits attributes from *GRLmodelElement* (see Figure 8).

Relationships

- Inherits relationships from *GRLmodelElement*.
- Contained by *GRLspec* (1): An *ElementLink* is contained in the GRL specification (see Figure 7).
- 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*.

Constraints

a. Inherits constraints from *GRLmodelElement*.

b. All instances of *ElementLink* must appear in one of its subclasses (that is, metaclass *ElementLink* is abstract).

c. The source and destination GRL linkable elements must be different.

b) Concrete grammar

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

Relationships

- Inherits relationships from *GRLmodelElement* (see Figure 8).
- Association with *LinkRef* (0..*): An *ElementLink* may have link references (see Figure 25).

c) Semantics

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

7.4.2 Contribution

A *Contribution* link describes how a source intentional element 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 (see Figure 9).

a) Abstract grammar

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.

Relationships

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

Constraints

a. Inherits constraints from *ElementLink*.

b. Actor definitions can neither be the source nor the destination of a contribution.

c. The destination intentional element (dest) must not be a resource or a belief.

d. quantitativeContribution ≥ –100 and quantitativeContribution ≤ 100.

b) Concrete grammar

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

Relationships

- Inherits relationships from *ElementLink*.

c) Semantics

A *Contribution* defines the level of impact that the satisfaction of a source intentional element 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.

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.

Semantic variations

The following paragraph is non-normative.

Modellers may impose additional stylistic constraints on the well-formedness of contributions. For example, as contributions links can be qualitative and partial, while goals, tasks and resources represent clear-cut concepts, a constraint may specify that intentional elements can only contribute to softgoals:

e. The destination intentional element must be a softgoal.

d) Model

None.

e) Example

Figure 14 is a GRL diagram that shows three contributions and two correlations linking five intentional elements.

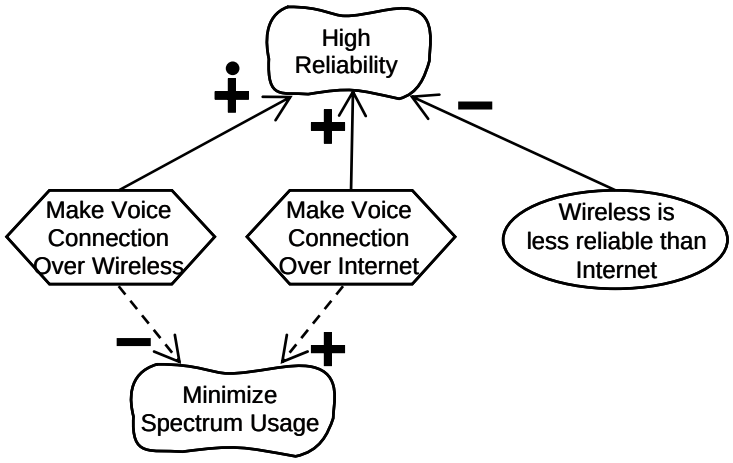


Figure 14 – Example: GRL contributions and correlations

- "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 7.6.7 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 9).

a) Abstract grammar

Attributes

- None (enumeration metaclass).

Relationships

- Used by *Contribution*.

Constraints

None.

b) Concrete grammar

Figure 15 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 7.6.7.

Figure 15 – Symbols: GRL contribution types

c) Semantics

The qualitative contribution of a source intentional element 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 (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 9).

The dependum specifies *what* the dependency is about, i.e., the intentional element around which a dependency relationship centers. With an intentional element as a source of the dependency, the depender may specify *why* it depends on the depender for the dependum. With an intentional element as a target of the dependency, the dependee may specify *how* it is required to provide or satisfy the dependum.

a) Abstract grammar

Attributes

- Inherits attributes from *ElementLink*.

Relationships

- Inherits relationships from *ElementLink*.

Constraints

a. Inherits constraints from *ElementLink*.

b. Belief intentional elements can neither be the source nor the destination of a dependency.

c. At least one of the GRL linkable elements linked by the dependency must be an actor definition or an intentional element contained in an actor definition.

d. If the source and destination linked by the dependency are intentional elements, then these intentional elements must 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.

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 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 *IntentionalElement*. *Why* is inside *Depender*, *How* is inside *Dependee*, and *What* is not inside any actor definition. A named arrow (—*D*→) indicates the presence of a dependency link *D* between the source and target GRL linkable elements involved.

1. *Depender* —*D1*→ *What* —*D2*→ *Dependee*
    - *Depender* depends on *Dependee* for *What*. *What* represents the dependum.
2. *Depender* —*D1*→ *How*
    - *Depender* depends on *Dependee* for *How*. The dependum is unknown.
3. *Why* —*D1*→ *What* —*D2*→ *Dependee*
    - *Why* in *Depender* depends on *Dependee* for *What*. *What* represents the dependum.
4. *Why* —*D1*→ *How*
    - *Why* in *Depender* depends on *Dependee* for *How*. The dependum is unknown.
5. *Depender* —*D1*→ *Dependee*
    - *Depender* depends on *Dependee*. The dependum is unknown.
6. *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.

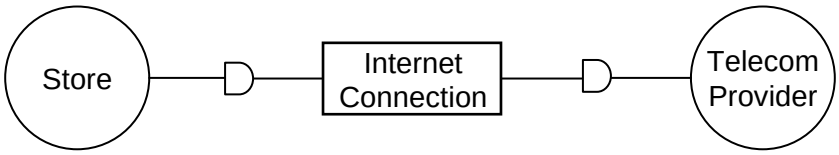


Figure 16 – Example: GRL dependencies (configuration 1)

1. *The Store depends on the Telecom Provider to provide an Internet Connection* (Figure 16). This is a configuration that focuses solely on strategic dependencies between actors. Why and how the dependum is provided are unknown.

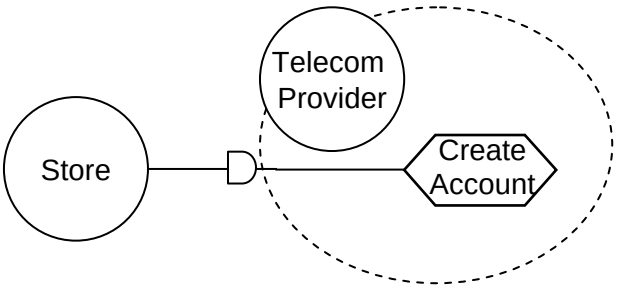


Figure 17 – Example: GRL dependencies (configuration 2)

2. *The Store depends on the Telecom Provider to create an account* (Figure 17). The dependum and why it is required are unknown.

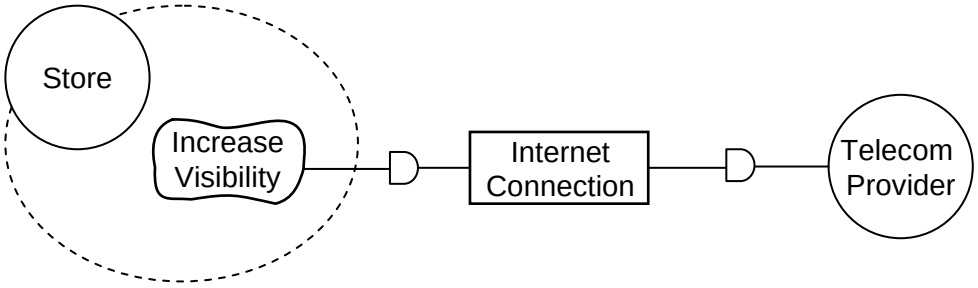


Figure 18 – Example: GRL dependencies (configuration 3)

3. *To increase its visibility, the Store depends on the Telecom Provider to provide an Internet Connection* (Figure 18). How the dependum is provided is unknown.

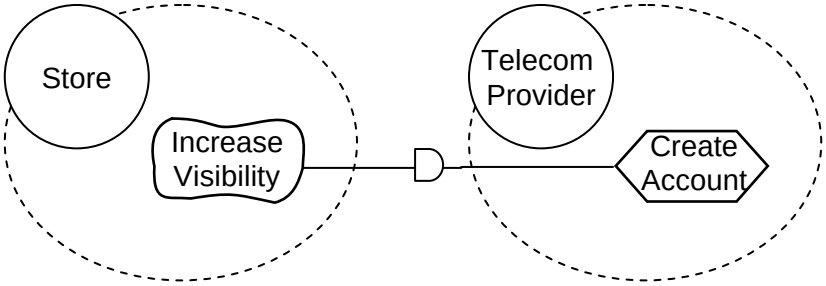


Figure 19 – Example: GRL dependencies (configuration 4)

4. *To increase its visibility, the Store depends on the Telecom Provider to create an account* (Figure 19). The dependum is unknown.

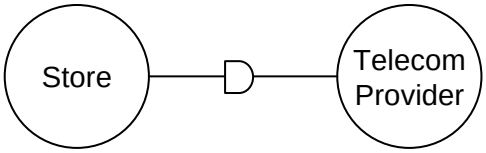


Figure 20 – Example: GRL dependencies (configuration 5)

5. *The Store depends on the Telecom Provider* (Figure 20). 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.

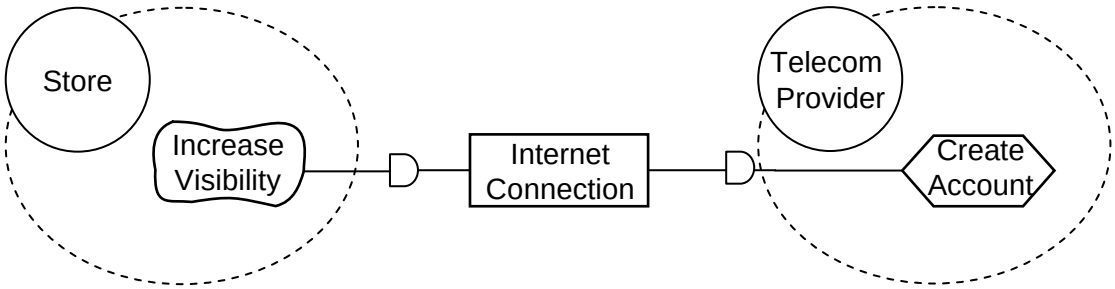


Figure 21 – Example: GRL dependencies (configuration 6)

6. *To increase its visibility, the Store depends on the Telecom Provider to provide an Internet Connection by creating an account* (Figure 21). 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 9).

a) Abstract grammar

Attributes

- Inherits attributes from *ElementLink*.

Relationships

- Inherits relationships from *ElementLink*.

Constraints

a. Inherits constraints from *ElementLink*.

b. Actor definitions can neither be the source nor the destination of a decomposition.

c. Intentional elements of type Belief can neither be the source nor 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.

Relationships

- Inherits relationships from *ElementLink*.

c) Semantics

Decomposition links connect the essential parts of an intentional element, which include subtasks that must be performed, subgoals that must be achieved, resources that must be accessible, and softgoals that must 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 enable 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.

Semantic variations

The following paragraph is non-normative.

Modellers may impose additional stylistic constraints on the well-formedness of decomposition links. For example, tasks could be limited to AND decomposition:

d. If the target of a decomposition link is a task intentional element, then its decomposition type must be AND.

d) Model

None.

e) Examples

Figure 22 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 7.6.7 for the details of the concrete syntax.

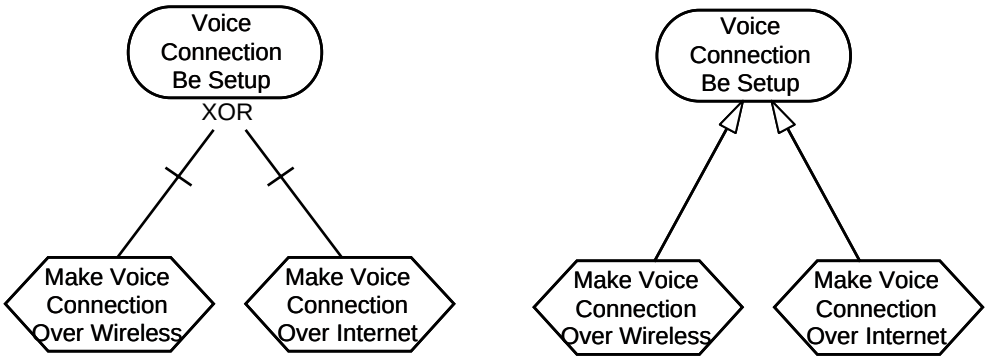


Figure 22 – Example: GRL XOR decomposition: normal (left) and means-end (right) presentations

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 9).

Attributes

- None (enumeration metaclass).

Relationships

- Used by *IntentionalElement*.

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 7.6.7.

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 intentional elements in a GRL model. These evaluation values, which can be quantitative or qualitative, are satisfaction levels that can then be propagated to the other intentional elements 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 amongst 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.

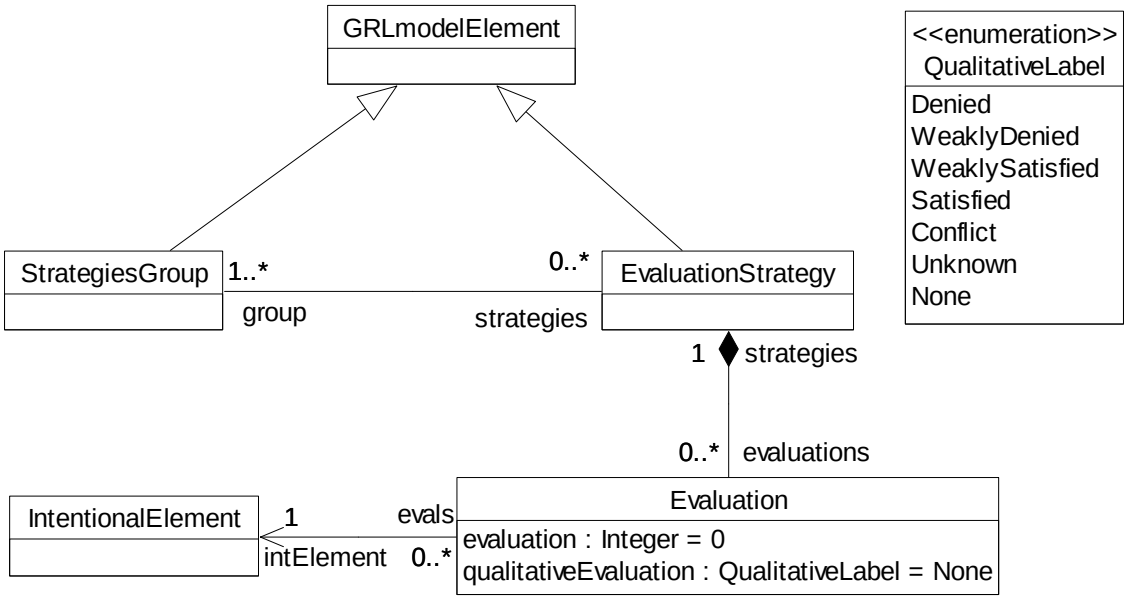


Figure 23 – Abstract grammar: GRL evaluation strategies

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 23).

a) Abstract grammar

Attributes

- Inherits attributes from *GRLmodelElement*.

Relationships

- Inherits relationships from *GRLmodelElement*.
- Contained by *GRLspec* (1): A *StrategiesGroup* is contained in the GRL specification (see Figure 7).
- Association with *EvaluationStrategy* (0..*): A *StrategiesGroup* may refer to evaluation strategies.

Constraints

a. Inherits constraints from *GRLmodelElement*.

b) Concrete grammar

A *StrategiesGroup* does not have a visual representation.

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 23).

a) Abstract grammar

Attributes

- Inherits attributes from *GRLmodelElement*.

Relationships

- Inherits relationships from *GRLmodelElement*.
- Contained by *GRLspec* (1): An *EvaluationStrategy* is contained in the GRL specification (see Figure 7).
- Association with *StrategiesGroup* (1..*): An *EvaluationStrategy* is referenced by at least one group of strategies.
- Composition of *Evaluation* (0..*): An *EvaluationStrategy* may contain evaluations.

Constraints

a. Inherits constraints from *GRLmodelElement*.

b) Concrete grammar

An *EvaluationStrategy* does not have a visual representation.

Relationships

- Inherits relationships from *GRLmodelElement*.

c) Semantics

The *Evaluation*s 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 that do not have any initial evaluation value, through the element links that connect them.

7.5.3 Evaluation

An *Evaluation* provides initial quantitative and qualitative evaluation values to an intentional element (see Figure 23).

a) Abstract grammar

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.

Relationships

- Contained by *EvaluationStrategy* (1): An *Evaluation* is contained in one evaluation strategy.
- Association with *IntentionalElement* (1): An *Evaluation* provides initial evaluation values to one intentional element definition.
- Uses *QualitativeLabel* enumeration.

Constraints

a. evaluation ≥ –100 and evaluation ≤ 100.

b) Concrete grammar

An *Evaluation* does not have a visual representation. However, it may impact the presentation of intentional element references (see clause 7.6.5)

c) Semantics

An *Evaluation* defines the initial level of satisfaction of an intentional element. 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.

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. It can be one of the following values: Denied, WeaklyDenied, WeaklySatisfied, Satisfied, Conflict, Unknown, and None (see Figure 23).

a) Abstract grammar

Attributes

- None (enumeration metaclass).

Relationships

- Used by *Evaluation*.

Constraints

None.

b) Concrete grammar

Figure 24 lists the icons that are used to annotate GRL intentional elements according to their (qualitative) satisfaction level for a given strategy evaluation. See usage in *IntentionalElementRef*, clause 7.6.5.

Figure 24 – Symbols: GRL qualitative labels

c) Semantics

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

- Denied: The intentional element is sufficiently dissatisfied.
- WeaklyDenied: The intentional element is partially dissatisfied.
- WeaklySatisfied: The intentional element is partially satisfied.
- Satisfied: The intentional element is sufficiently satisfied.
- Conflict: There are arguments strongly in favour and strongly against the satisfaction of the intentional element.
- Unknown: The satisfaction level of the intentional element is unknown.
- None: The intentional element 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. This makes GRL diagrams less crowded, without loss of information.

7.6 GRL concrete grammar metaclasses

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

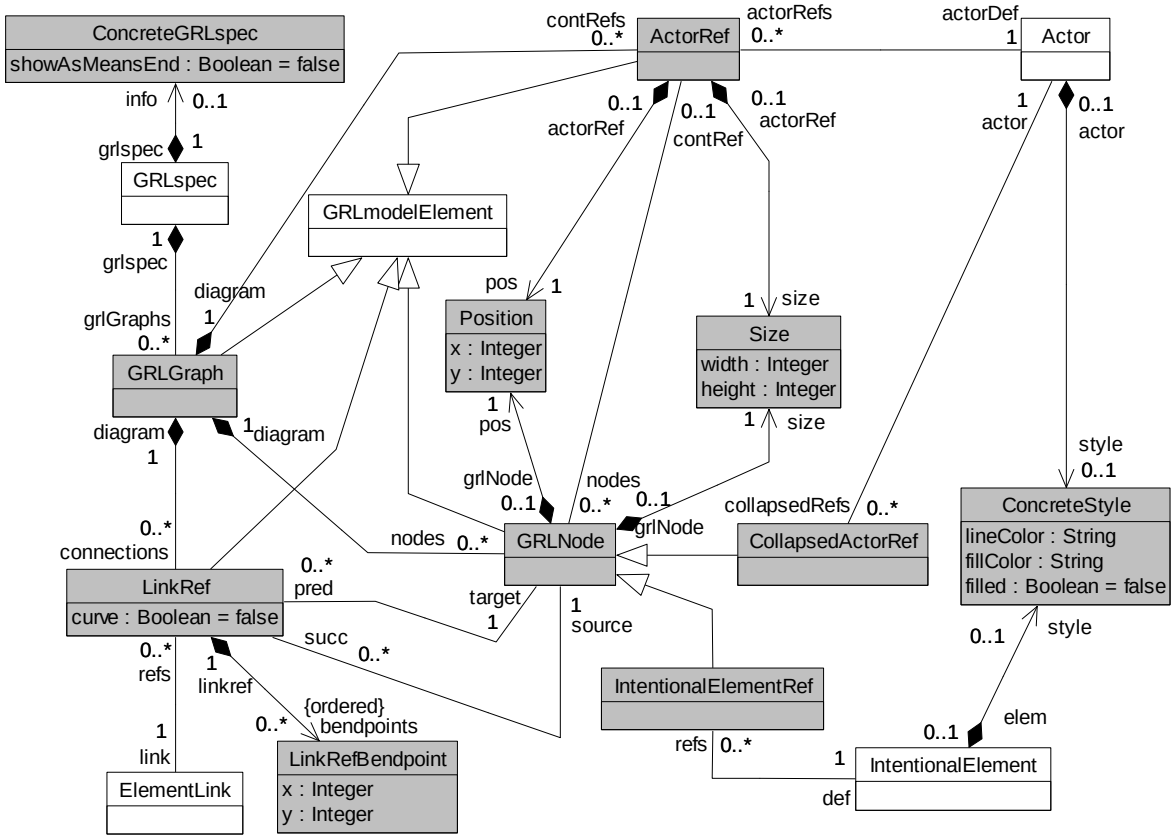


Figure 25 – Concrete grammar: GRL concrete syntax metaclasses

7.6.1 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 25).

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 7.6.7).

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.

7.6.2 GRLGraph

A *GRLGraph* is a container for all actor references, GRL nodes (collapsed actor references and intentional 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 25).

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 *Comment*s.

Attributes

- Inherits attributes from *GRLmodelElement*.

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 50).

7.6.3 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 25).

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 26, where the name of the actor reference is ActorRef). The line and fill colors of the actor reference are those of the actor definition's concrete style (*ConcreteStyle*).

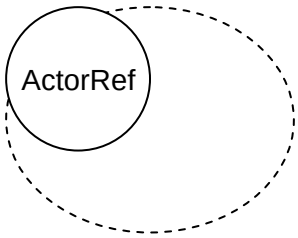


Figure 26 – Symbol: GRL actor reference

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 and quantitativeVal) are discussed in clause 11.1. Depending on the nature of the analysis, a qualitative label icon (i) or a quantitative integer value in parentheses (ii) can be used to annotate the actor reference symbol (see Figure 27). The icons for the qualitative labels are defined in clause 7.5.4 (*QualitativeLabel*).

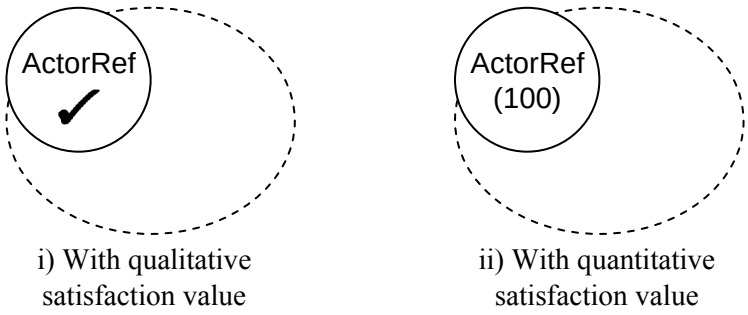


Figure 27 – Example: GRL actors with satisfaction values

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 28. The *Label* is not used in this presentation (see clause 7.6.3, numeral d) for label usage). The same layout principles apply also to *IntentionalElementRef*s and *CollapsedActorRef*s.

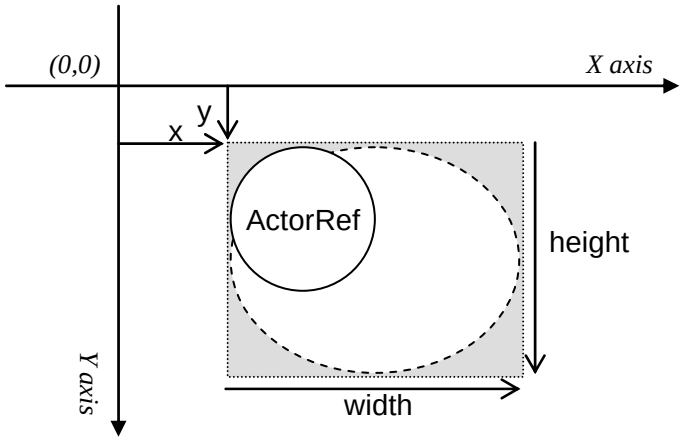


Figure 28 – Layout: Position and size of ActorRef, IntentionalElementRef, and CollapsedActorRef

Attributes

- Inherits attributes from *GRLmodelElement*.

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 46).
- Association with *Actor* (1): An *ActorRef* refers to one actor definition.

- Association with *GRLNode* (0..*): An *ActorRef* may include GRL nodes.

Constraints

a. Inherits constraints from *GRLmodelElement*.

b. The name of an *ActorRef* is the same as the name of its referenced *Actor* definition.

c. The name of an *ActorRef* including its annotations is inside the actor symbol.

d. Rectangles containing the actor symbol and the actor boundary symbol share the same top-left corner.

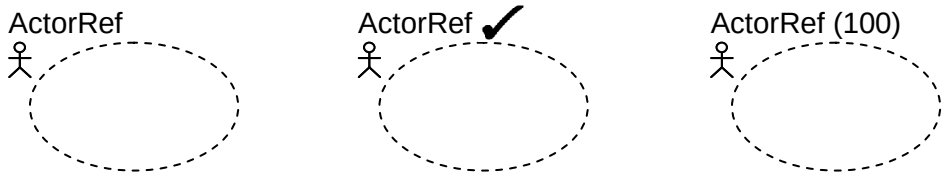
e. The boundary of an *ActorRef* must not overlap with the boundary of another *ActorRef*.

c) Semantics

None.

d) Model

An alternate way of displaying an *ActorRef*, illustrated in Figure 29, 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 resulting from the analysis of an *EvaluationStrategy*.



i) Alternative presentation for an actor reference

ii) With qualitative satisfaction value

iii) With quantitative satisfaction value

Figure 29 – 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 79) for an illustration of these layout principles.

7.6.4 GRLNode

GRLNode is an abstraction of intentional element references and collapsed actor references in a GRL diagram. GRL nodes can be included in actor references and they have a position and a size (see Figure 25).

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.

Attributes

- Inherits attributes from *GRLmodelElement*.

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*.

Constraints

a. Inherits constraints from *GRLmodelElement*.

b. All instances of *GRLNode* must appear in one of its subclasses (that is, metaclass *GRLNode* is abstract).

c. The *GRLGraph* that contains the *GRLNode* must be the *GRLGraph* that contains *LinkRef*s associated as pred.

d. The *GRLGraph* that contains the *GRLNode* must be the *GRLGraph* that contains *LinkRef*s associated as succ.

e. If the *GRLNode* is included in one *ActorRef*, then the *GRLGraph* that contains this GRL node must be the *GRLGraph* that contains this actor reference.

f. If the *GRLNode* is included in one *ActorRef*, then the position and size of this GRL node must be such that the node is entirely contained inside the boundary of the actor reference.

7.6.5 IntentionalElementRef

An intentional element reference (*IntentionalElementRef*) shows an intentional element on a GRL diagram. Its presentation depends on the type of the intentional element definition it refers to. In a URN specification, the same intentional element definition may be referenced many times in the same GRL diagram and in many GRL diagrams (see Figure 25).

a) Abstract grammar

None. This is a concrete syntax metaclass only.

b) Concrete grammar

Figure 30 lists the symbols used for 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.

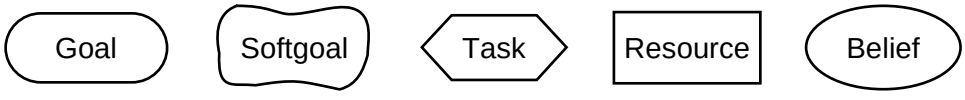


Figure 30 – Symbols: GRL intentional element references

The intentional element reference name (from superclass *URNmodelElement*) is displayed inside the symbol (in Figure 30, the type of each intentional element is used as its name, for illustration purpose). If the intentional element is contained in an actor, then the importance information may also be displayed following the name. If the importanceQualitative information is displayed, then (H) is used for High, (M) for Medium, and (L) for Low. None is not displayed. If the quantitative importance information is displayed, then the value is shown in parentheses, but only if greater than zero (see Figure 31).

Figure 31 – Example: GRL intentional elements with importance values

The line and fill colors of the intentional element reference are those of the intentional element definition's concrete style (*ConcreteStyle*).

When analysing an *EvaluationStrategy* in the GRL specification, the intentional element reference symbol can be supplemented with a symbol denoting the current evaluation value of the referenced intentional element definition (see Figure 32). The values to be reported in this way (qualitativeVal and quantitativeVal) 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 reference symbol. The icons for the qualitative labels are defined in clause 7.5.4 (*QualitativeLabel*).

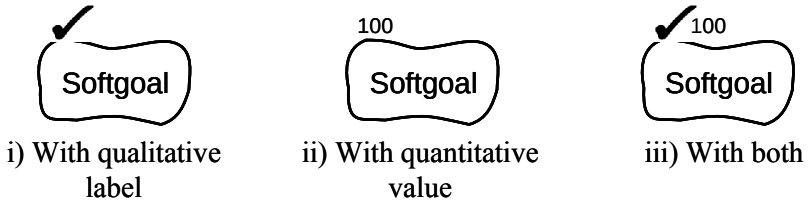


Figure 32 – Example: GRL intentional elements with satisfaction values

If the current *EvaluationStrategy* has an *Evaluation* for the referenced intentional element, then the current annotation is supplemented with a star (*), which indicates that this is an initial value for this strategy (see Figure 33).

Figure 33 – Example: GRL intentional elements 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 28. The additional annotations are added at a fixed position above the symbol, starting from the left.

Attributes

- Inherits attributes from *GRLNode*.

Relationships

- Inherits relationships from *GRLNode*.
- Association with *IntentionalElement* (1): An *IntentionalElementRef* references one intentional element definition.

Constraints

a. Inherits constraints from *GRLNode*.

b. The name of an *IntentionalElementRef* is the same as the name of its *IntentionalElement* definition.

c. If the *IntentionalElementRef* is included by an *ActorRef*, then the referenced *IntentionalElement* definition must be included by the referenced *Actor* definition.

d. The name of an *IntentionalElementRef* including its importance value is inside the intentional element symbol.

e. Intentional element symbols on the same GRL diagram must not overlap.

c) Semantics

None.

d) Model

The following clause is non-normative.

When evaluating a strategy in the GRL specification, the fill color of the intentional element symbol may be overridden temporarily to provide additional visual feedback about the satisfaction level of the referenced intentional element definition. For example, the following color scheme may 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.

7.6.6 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 25).

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 34, where the name of this reference is CollapsedActorRef). The line and fill colors of the collapsed actor reference are those of the actor definition's concrete style (*ConcreteStyle*).

Figure 34 – Symbol: GRL collapsed actor reference

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 and quantitativeVal) are discussed in clause 11.1. Depending on the nature of the analysis, a qualitative label icon (i) or a quantitative integer value between parentheses (ii) can be used to annotate the actor reference symbol (see Figure 35). The icons for the qualitative labels are defined in clause 7.5.4 (*QualitativeLabel*).

i) With qualitative satisfaction value

ii) With quantitative satisfaction value

Figure 35 – 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 28.

Attributes

- Inherits attributes from *GRLNode*.

Relationships

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

Constraints

a. Inherits constraints from *GRLNode*.

b. The name of a *CollapsedActorRef* is the same as the name of its referenced *Actor* definition.

c. The name of a *CollapsedActorRef* including its annotations is inside the collapsed actor symbol.

7.6.7 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 25).

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 36).

Figure 36 – 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 37).

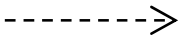
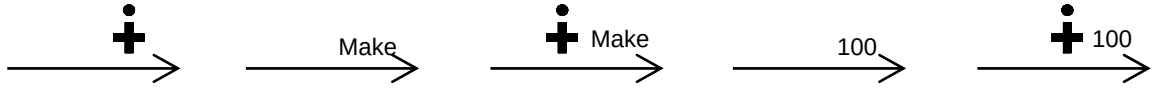


Figure 37 – Symbol: GRL correlation

- 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 38).



i) Icon only ii) Text only iii) Icon and text iv) Number only v) Icon and number

Figure 38 – Examples: GRL contribution links with contribution values

For a *Dependency*, the symbol is a line with a D on it (see Figure 39). The flat side of the D is pointed at the source. There is no *Label* associated with a dependency link.

Figure 39 – 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 in Figure 40:

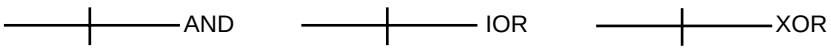


Figure 40 – Symbol: GRL decompositions

- 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 41). The presentation of the AND decomposition remains unchanged.

Figure 41 – 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.

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.

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 46).
- 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.

Constraints

a. Inherits constraints from *GRLmodelElement*.

b. The source and target GRL nodes must be different.

c. The *LinkRef* has a *Label* if and only if the *ElementLink* to which the *LinkRef* refers is a *Contribution*.

d. If the source GRL node is an *IntentionalElementRef*, then the *IntentionalElement* definition referenced by that source must be the source of the *ElementLink* to which the *LinkRef* refers.

e. If the target GRL node is an *IntentionalElementRef*, then the *IntentionalElement* definition referenced by that target must be the destination of the *ElementLink* to which the *LinkRef* refers.

f. If the source GRL node is a *CollapsedActorRef*, then the *Actor* definition referenced by that source must be the source of the *ElementLink* to which the *LinkRef* refers.

g. If the target GRL node is a *CollapsedActorRef*, then the *Actor* definition referenced by that target must be the destination of the *ElementLink* to which the *LinkRef* refers.

h. 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 42). There is no semantic distinction between a non-filled and a filled D.

Figure 42 – 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 43).

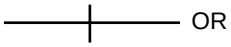


Figure 43 – Symbol: Alternative presentation for IOR decomposition

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 bendpoints. If the value of the curve attribute is false, then straight line segments are used (Figure 44).

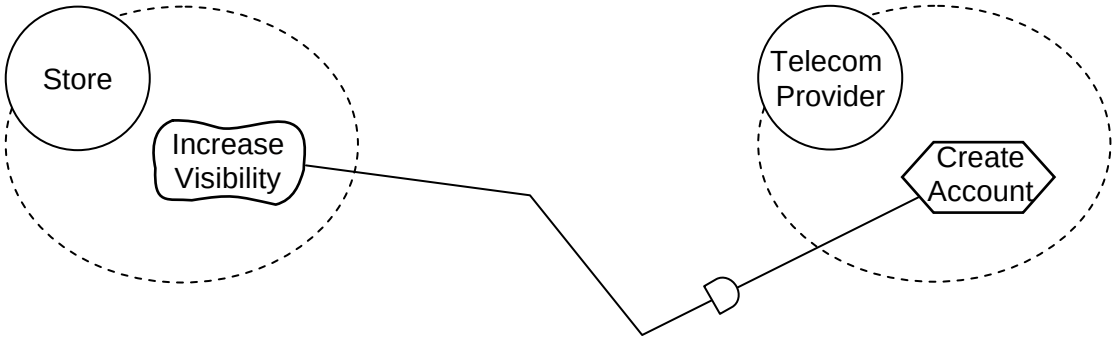


Figure 44 – Example: GRL link with two bend points shown with straight lines

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 45).

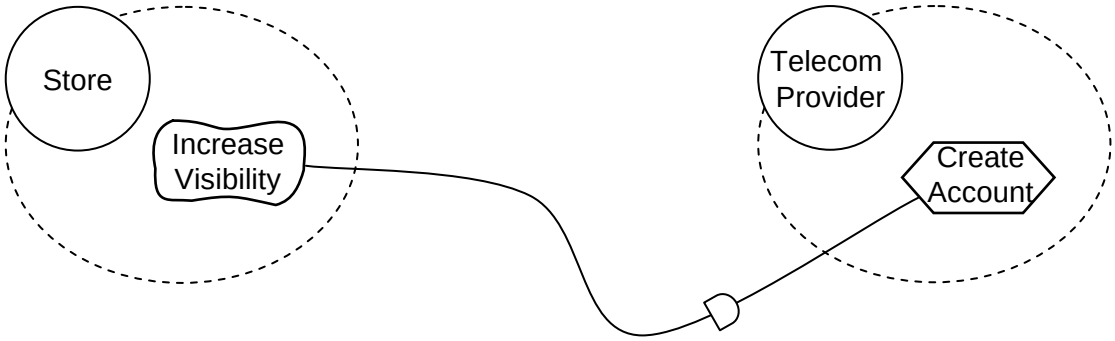


Figure 45 – Example: GRL link with two bend points shown as a curved line

7.6.8 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*. It 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, 7.6.7, and 8.2.3, respectively) (see Figure 46).

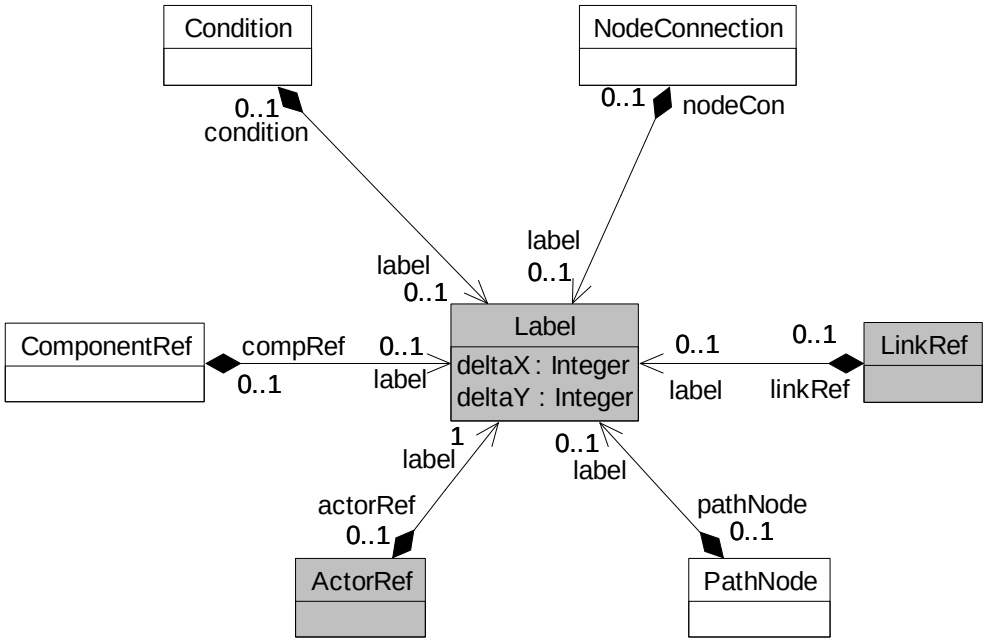


Figure 46 – Concrete grammar: Label metaclass

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.

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).
- 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).

Constraints

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

7.6.9 LinkRefBendpoint

A bend point is a fixed point on a GRL diagram through which a link reference must 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 25).

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 7.6.7).

Attributes

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

Constraints

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

7.6.10 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 47).

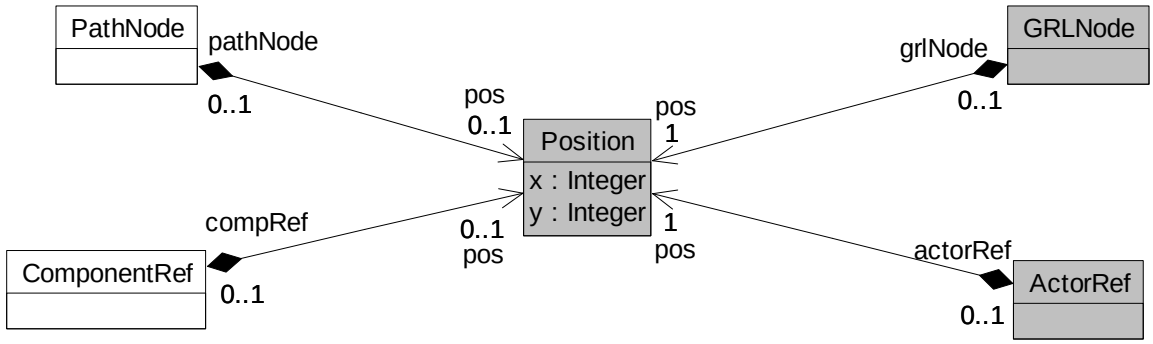


Figure 47 – Concrete grammar: Position metaclass

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.

Attributes

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

Constraints

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

7.6.11 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 48).

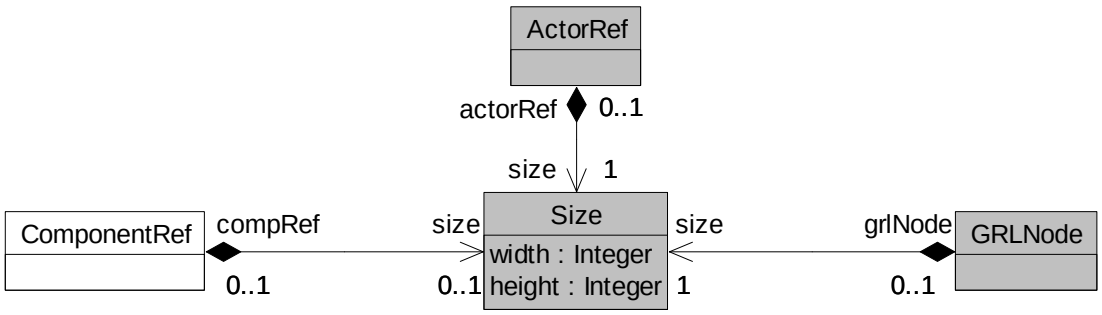


Figure 48 – Concrete grammar: Size metaclass

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.

Attributes

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

Constraints

a. width > 0

b. height > 0

c. Each *Size* instance is contained in exactly one instance of type *ComponentRef*, *ActorRef*, or *GRLNode*.

7.6.12 ConcreteStyle

The *ConcreteStyle* metaclass is used to specify the color of various graphical elements in GRL and UCM diagrams. A *ConcreteStyle* is contained by an *Actor*, *IntentionalElement*, or *Component*. This information is attached to the definitions of intentional elements, actor definitions, and component definitions so their references can be colored consistently across diagrams (see Figure 49).

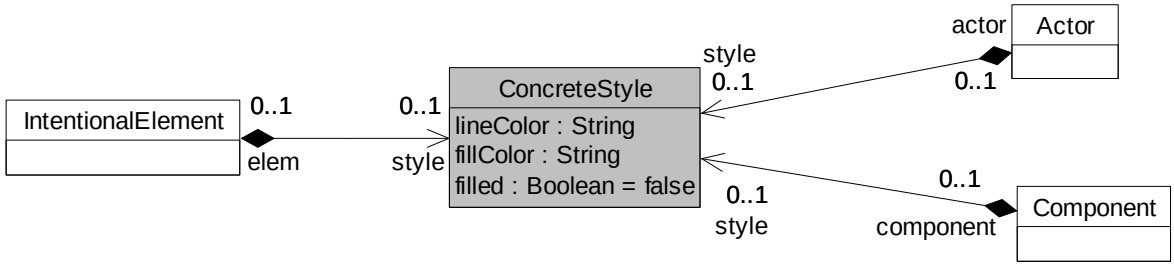


Figure 49 – Concrete grammar: ConcreteStyle metaclass

a) Abstract grammar

None. This is a concrete syntax metaclass only.

b) Concrete grammar

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

Colors are represented as a string using a red, green, and blue color model. The intensity of each color 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 color yellow (red=255, green=255, blue=0).

Attributes

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

Constraints

a. Each *ConcreteStyle* instance is contained in exactly one instance of type *Actor*, *IntentionalElement*, or *Component*.

7.6.13 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 50).

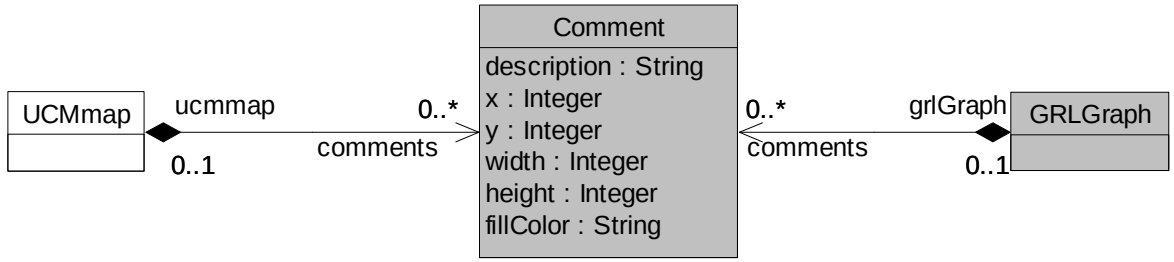


Figure 50 – Concrete grammar: Comment metaclass

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 51). The description text is wrapped on multiple lines according to the width.

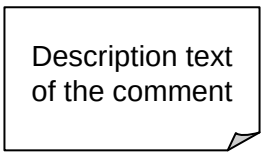


Figure 51 – Symbol: URN comment

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).

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.
- y (*Integer*): Vertical coordinate (on the Y axis) of the *Comment*, in point units.
- width (*Integer*): Width of the *Comment*, in point units.
- height (*Integer*): Height of the *Comment*, in point units.
- fillColor (String): Fill color of the *Comment*. The color conventions of *ConcreteStyle* apply.

Constraints

a. width > 0

b. height > 0

c. Each *Comment* instance is contained in exactly one instance of type *UCMmap* or *GRLGraph*.

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 7.6. 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 52).

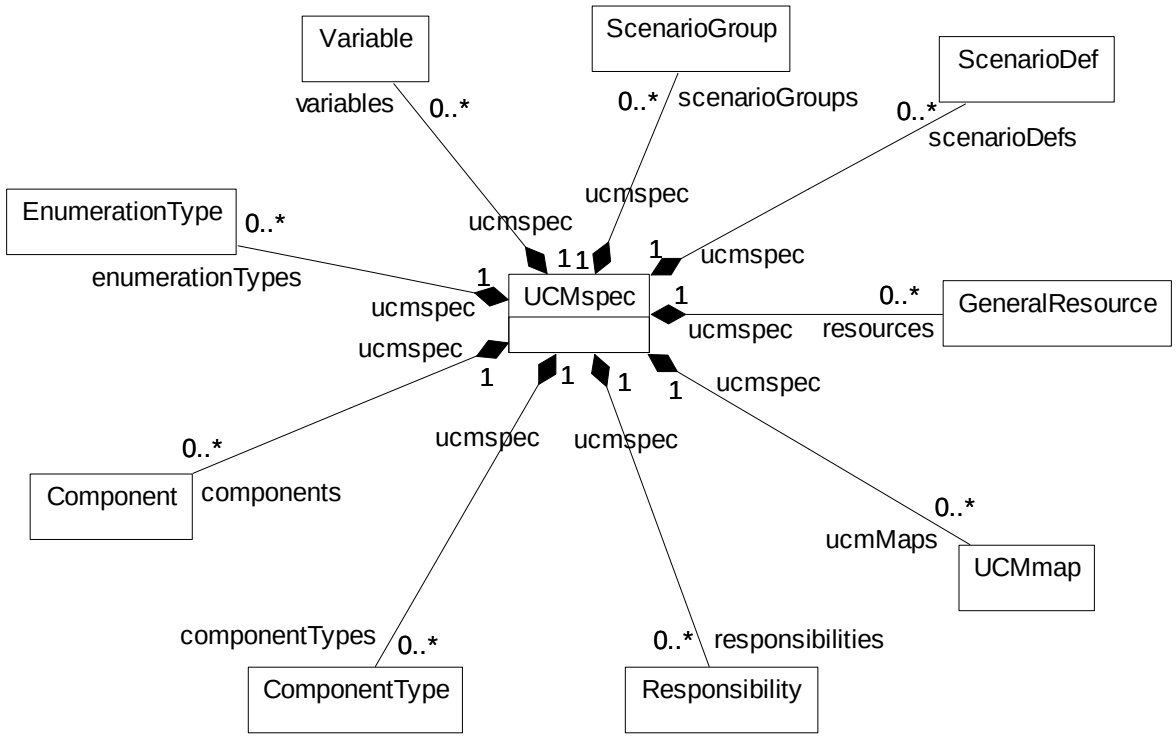


Figure 52 – Abstract grammar: UCM specification

a) Abstract grammar

Attributes

None.

Relationships

- Contained by *URNspec* (1): The *UCMspec* is contained in the URN specification (see Figure 2).
- 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.

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 53).

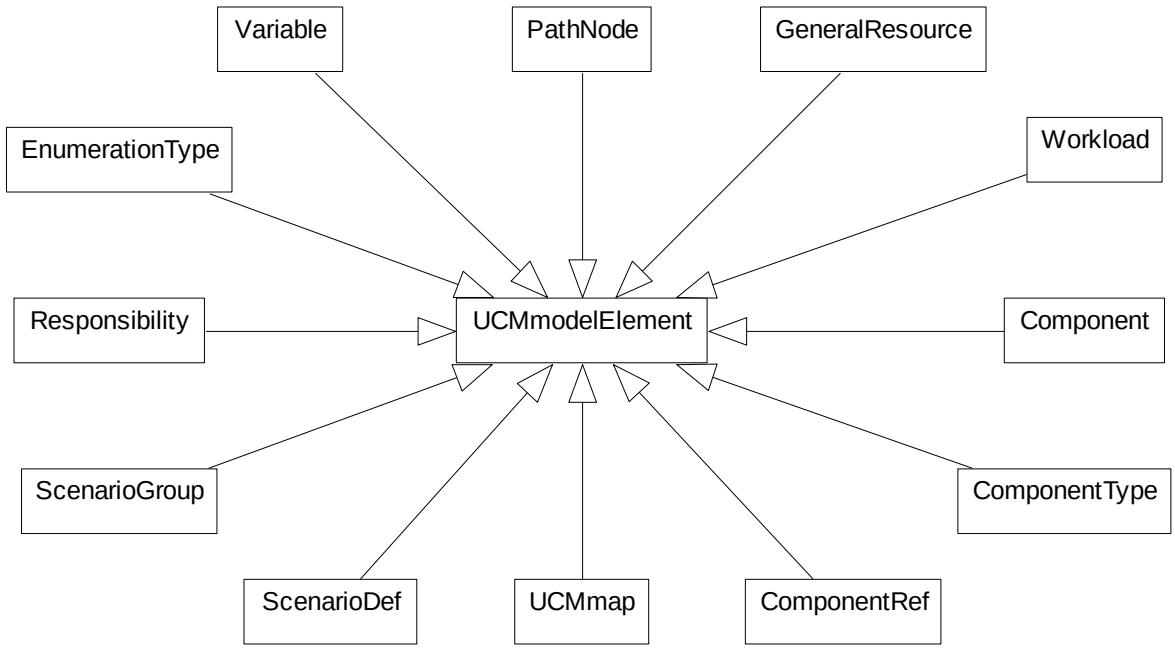


Figure 53 – Abstract grammar: UCM model elements

a) Abstract grammar

Attributes

- Inherits attributes from *URNmodelElement* (see Figure 2).

Relationships

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

Constraints

a. Inherits constraints from *URNmodelElement*.

b. All instances of *UCMmodelElement* must 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.

Relationships

- Inherits relationships from *URNmodelElement* (see Figure 2).

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

*UCMmap*s and *PathNode*s 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 54).

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

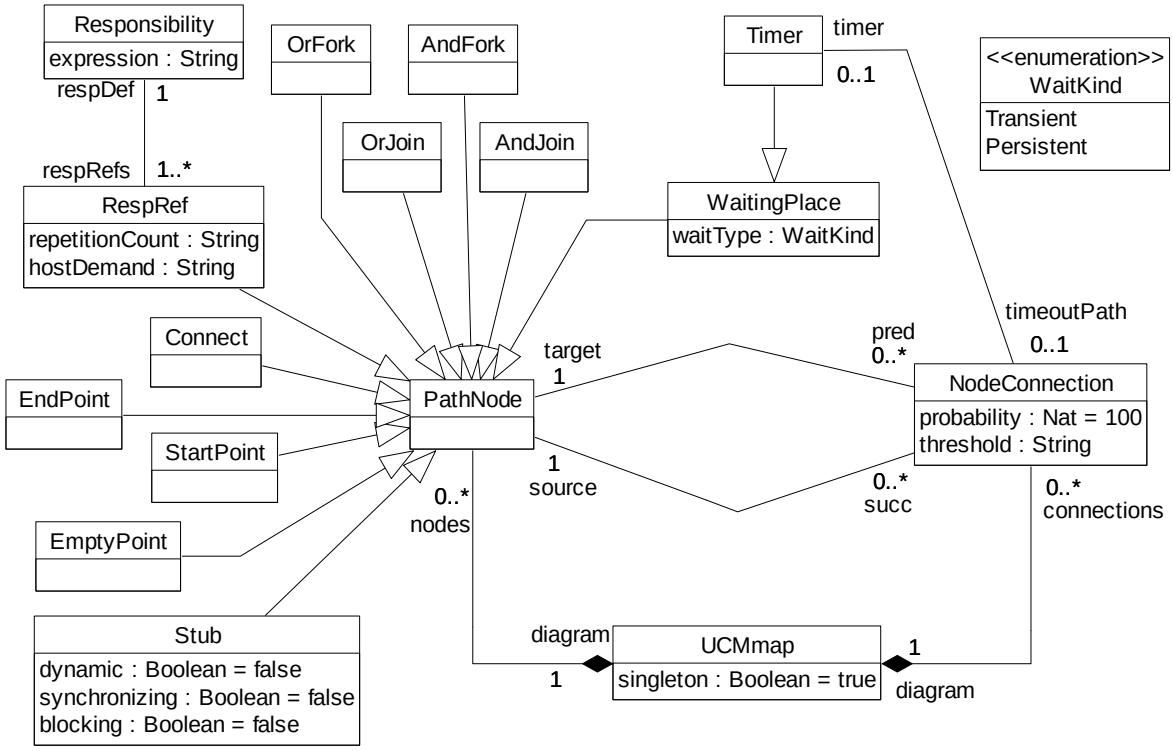


Figure 54 – Abstract grammar: UCM paths and path nodes

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 instance of it exists in the UCM specification. A map may be reused as a plug-in map for stubs (see Figure 54, Figure 68, and Figure 76).

a) Abstract grammar

Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 53).
- singleton (Boolean): Indicates whether one (true) or more (false) runtime instances of a *UCMmap* may exist in the UCM specification. Default value is true (i.e., only one instance may exist).

Relationships

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *UCMmap* is contained in the UCM specification (see Figure 52).
- 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.

Constraints

a. Inherits constraints from *UCMmodelElement*.

b) Concrete grammar

UCMmap represents the map diagram and as such has no concrete syntax except for *Comment*s.

Relationships

- Inherits relationships from *UCMmodelElement* (see Figure 53).
- Composition of *Comment* (0..*): A *UCMmap* may have comments (see Figure 50).

c) Semantics

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

d) Model

None.

e) Examples

The UCM diagrams in Figure 55 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:

- OCS (originating call screening) is an originating user feature that filters outgoing calls to phone numbers on a screening list (see Figure 55.iv).
- TL (teen line) 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 55.vi).
- TCS (terminating call screening) is a terminating user feature that filters incoming calls from phone numbers on a screening list (see Figure 55.v).

The basic behaviour of the simple phone system is defined on the root map (Simple Connection) of the UCM model (see Figure 55.i) and the root map's two plug-in maps Originating Features and Terminating Features (see Figure 55.ii and Figure 55.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 55.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 55.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.

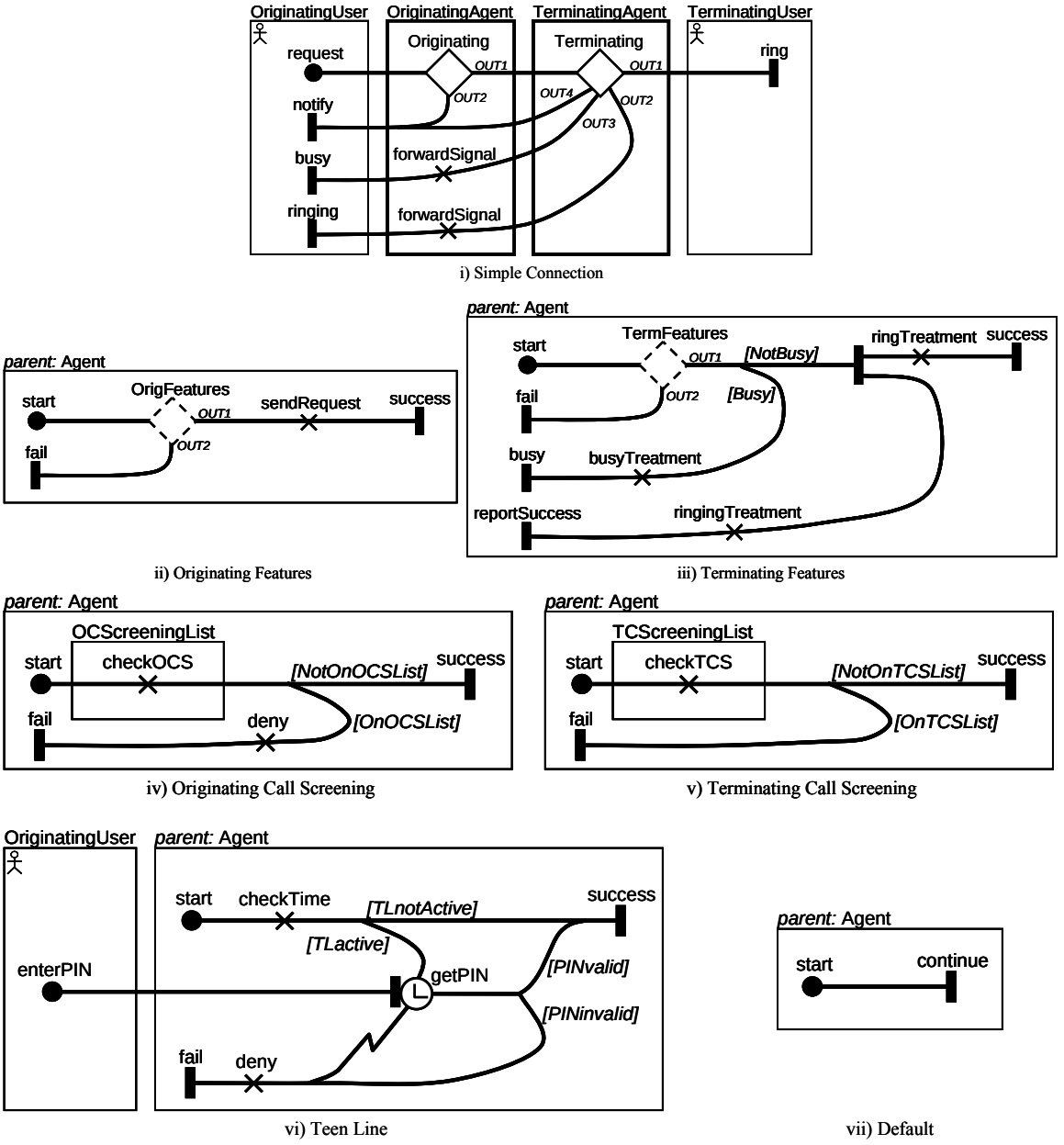


Figure 55 – Example: UCM model

The plug-in bindings of the UCM model in Figure 55 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 1. 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 dialed 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 1 – Example: Plug-in bindings for the UCM model

Stub	*Plug-in Map*	*In-path / Start Point*	*Out-path / EndPoint*	*Precondition*
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 Figure 54 and Figure 76).

a) Abstract grammar

Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 53).

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*, *OrFork*, *AndFork*, *OrJoin*, *AndJoin*, *Stub*, *EndPoint*, *StartPoint*, *EmptyPoint*, and *Connect*.

Constraints

a. Inherits constraints from *UCMmodelElement*.

b. All instances of *PathNode* must appear in one of its subclasses (that is, metaclass *PathNode* is abstract).

c. The *UCMmap* that contains the *PathNode* must be the *UCMmap* that contains *NodeConnection*s associated as pred.

d. The *UCMmap* that contains the *PathNode* must be the *UCMmap* that contains *NodeConnection*s associated as succ.

e. If the *PathNode* is included in one *ComponentRef*, then the *UCMmap* that contains this *PathNode* must 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 center of the *PathNode* is indicated by its *Position* (x, y). The bottom-center 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-center 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 56) for an illustration of these layout principles.

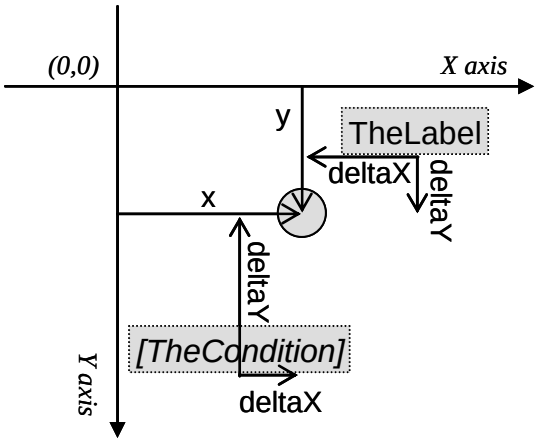


Figure 56 – Layout: Position, label, and condition for PathNode

Relationships

- Inherits relationships from *UCMmodelElement* (see Figure 53).
- Composition of *Label* (0..1): A *PathNode* may have one label (see Figure 46).
- Composition of *Position* (0..1): A *PathNode* may have one position (see Figure 47).
- *PathNode* is a superclass of *DirectionArrow*.

Constraints

f. If the *PathNode* is bound to one *ComponentRef*, then the *Position* of this *PathNode* must be such that the node is entirely contained inside the boundary of the *ComponentRef*.

g. 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 Figure 54 and Figure 68).

a) Abstract grammar

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.

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 a timer.

Constraints

a. probability ≥ 0 and probability ≤ 100.

b. The value of probability may be less than 100 only for a *NodeConnection* with a source path node of type *OrFork* or *Timer*.

c. The threshold must be empty or an Integer expression, as defined in clause 9.3.

d. The threshold must evaluate to a non-negative Integer value, or it may be empty, in which case it is deemed to evaluate to 0.

e. 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.

f. The source *PathNode* of a *NodeConnection* must be different from the target *PathNode* of the same *NodeConnection*.

g. A *NodeConnection* may have one *Condition* only if the source path node of the *NodeConnection* is of type *OrFork*, *AndFork*, *AndJoin*, or *WaitingPlace*.

h. A *NodeConnection* may have an *InBinding* only if the target path node of the *NodeConnection* is of type *Stub*.

i. A *NodeConnection* may have an *OutBinding* only if the source path node of the *NodeConnection* is of type *Stub*.

j. 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 *PathNode*s.

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-center 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-center 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).

Relationships

- Composition of *Label* (0..1): A *NodeConnection* may have one label (see Figure 46).

Constraints

k. 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 *PathNode*s linked by *NodeConnection*s 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 percent 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 only sense 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 must be fulfilled (i.e., must 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 only sense for OR-forks, AND-forks, AND-joins, 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 *InBinding*s and *OutBinding*s, 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 54).

a) Abstract grammar

Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 53).
- 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.

Relationships

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *Responsibility* definition is contained in the UCM specification (see Figure 52).
- Composition of *Demand* (0..*): A *Responsibility* definition may contain demands (see Figure 85).

- Association with *RespRef* (1..*): A *Responsibility* definition may be the definition for at least one responsibility reference.

Constraints

a. Inherits constraints from *UCMmodelElement*.

b. Any two *Responsibility* definitions cannot share the same name inside a URN specification.

c. The name of a *Responsibility* definition cannot be an empty String.

d. The expression must be an action, as defined in clause 9.4.

b) Concrete grammar

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

Relationships

- Inherits relationships from *UCMmodelElement* (see Figure 53).

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 *Variable*s. 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 *Demand*s on processing resources (see clause 8.6.15).

8.2.5 RespRef

RespRef is a path node that references a responsibility definition (see Figure 54).

a) Abstract grammar

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.

Relationships

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

Constraints

a. Inherits constraints from *PathNode*.

b. The name of a *RespRef* must be the same as the name of its associated *Responsibility* definition.

c. The repetitionCount must be empty or an Integer expression, as defined in clause 9.3.

d. The repetitionCount must evaluate to a positive Integer value or it may be empty, in which case it is deemed to evaluate to 1.

e. The hostDemand must be an Integer expression, as defined in clause 9.3.

f. The hostDemand must evaluate to a non-negative Integer value.

g. A *RespRef* is the source *PathNode* of exactly one *NodeConnection*.

h. 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 57).

Figure 57 – Symbol: UCM responsibility reference

Relationships

- Inherits relationships from *PathNode*.

Constraints

i. Inherits constraints from *PathNode*.

j. A *RespRef* must have one *Label*.

c) Semantics

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

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 *StartPoint* may also trigger scenario definitions (see Figure 54, Figure 68 and Figure 83).

a) Abstract grammar

Attributes

- Inherits attributes from *PathNode*.

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 85).
- Association with *InBinding* (0..*): A *StartPoint* may have in-bindings.

Constraints

a. Inherits constraints from *PathNode*.

b. A *StartPoint* is the source *PathNode* of exactly one *NodeConnection*.

c. A *StartPoint* is the target *PathNode* of zero or one *NodeConnection*.

d. If a *StartPoint* is the target *PathNode* of one *NodeConnection*, the source *PathNode* of the *NodeConnection* is of type *Connect*.

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. 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 Figure 58 and Figure 56). If no label attribute is defined for the concrete condition, then the square brackets also do not need to be shown. The symbol of a start point is the same as the symbol of a *WaitingPlace*.

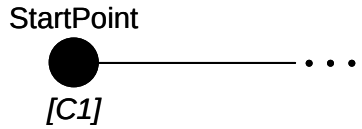


Figure 58 – Symbol: UCM start point

Relationships

- Inherits relationships from *PathNode*.

Constraints

e. 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 play a role in: a) the hierarchical structuring of UCM specifications through *InBinding*s, 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 *Workload*s, as explained in clause 8.6.

8.2.7 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 scenario definitions (see Figure 54, Figure 68, and Figure 83).

a) Abstract grammar

Attributes

- Inherits attributes from *PathNode*.

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.

Constraints

a. Inherits constraints from *PathNode*.

b. An *EndPoint* is the source *PathNode* of zero or one *NodeConnection*.

c. An *EndPoint* is the target *PathNode* of exactly one *NodeConnection*.

d. 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 Figure 59 and Figure 56). If no label attribute is defined for the concrete condition, then the square brackets also do not need to be shown.

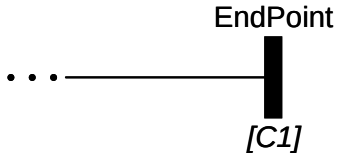


Figure 59 – Symbol: UCM end point

Relationships

- Inherits relationships from *PathNode*.

Constraints

e. 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 *OutBinding*s, as explained in clause 8.3; and b) UCM scenario definitions, as explained in clause 8.5.

8.2.8 OrFork

OrFork is a path node that represents a guarded choice point for alternative branches in scenario behaviour (see Figure 54).

a) Abstract grammar

Attributes

- Inherits attributes from *PathNode*.

Relationships

- Inherits relationships from *PathNode*.

Constraints

a. Inherits constraints from *PathNode*.

b. An *OrFork* is the source *PathNode* of one or more *NodeConnection*s.

c. 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 60).

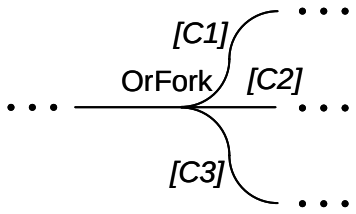


Figure 60 – Symbol: UCM OR-fork

Relationships

- Inherits relationships from *PathNode*.

Constraints

d. 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.

8.2.9 OrJoin

OrJoin is a path node that represents a merge point for alternative or concurrent branches in scenario behaviour (See Figure 54).

a) Abstract grammar

Attributes

- Inherits attributes from *PathNode*.

Relationships

- Inherits relationships from *PathNode*.

Constraints

a. Inherits constraints from *PathNode*.

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

c. An *OrJoin* is the target *PathNode* of one or more *NodeConnection*s.

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 61).

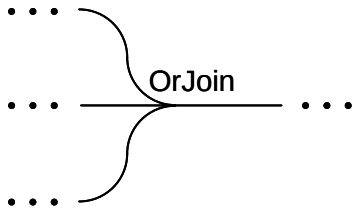


Figure 61 – Symbol: UCM OR-join

Relationships

- Inherits relationships from *PathNode*.

Constraints

d. 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.

8.2.10 AndFork

AndFork is a path node that represents the beginning of concurrent branches in scenario behaviour (see Figure 54).

a) Abstract grammar

Attributes

- Inherits attributes from *PathNode*.

Relationships

- Inherits relationships from *PathNode*.

Constraints

a. Inherits constraints from *PathNode*.

b. An *AndFork* is the source *PathNode* of one or more *NodeConnection*s.

c. 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 62).

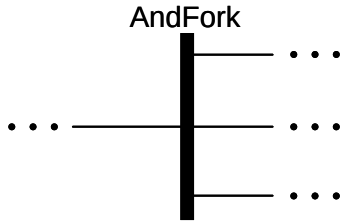


Figure 62 – Symbol: UCM AND-fork

Relationships

- Inherits relationships from *PathNode*.

Constraints

d. 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.

8.2.11 AndJoin

AndJoin is a path node that represents a synchronization point of alternative or concurrent paths in scenario behaviour (see Figure 54).

a) Abstract grammar

Attributes

- Inherits attributes from *PathNode*.

Relationships

- Inherits relationships from *PathNode*.

Constraints

a. Inherits constraints from *PathNode*.

b. An *AndJoin* is the source *PathNode* of exactly one *NodeConnection*.

c. An *AndJoin* is the target *PathNode* of one or more *NodeConnection*s.

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 63).

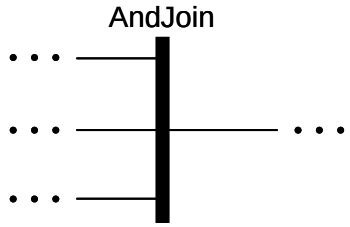


Figure 63 – Symbol: UCM AND-join

Relationships

- Inherits relationships from *PathNode*.

Constraints

d. 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.

8.2.12 EmptyPoint

EmptyPoint is a path node that is used to asynchronously connect two paths (see Figure 54).

a) Abstract grammar

Attributes

- Inherits attributes from *PathNode*.

Relationships

- Inherits relationships from *PathNode*.

Constraints

a. Inherits constraints from *PathNode*.

b. An *EmptyPoint* is the source *PathNode* of one or two *NodeConnection*s.

c. An *EmptyPoint* is the target *PathNode* of exactly one *NodeConnection*.

d. If an *EmptyPoint* is the source *PathNode* of two *NodeConnection*s, the target *PathNode* of exactly one of the two *NodeConnection*s 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 64).

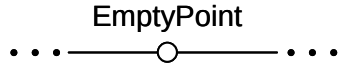


Figure 64 – Symbol: UCM empty point

Relationships

- Inherits relationships from *PathNode*.

Constraints

e. 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 *NodeConnection*s 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.

8.2.13 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 54).

a) Abstract grammar

Attributes

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

Relationships

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

Constraints

a. Inherits constraints from *PathNode*.

b. A *WaitingPlace* that is not a *Timer* is the source *PathNode* of exactly one *NodeConnection*.

c. A *WaitingPlace* is the target *PathNode* of one or two *NodeConnection*s.

d. If a *WaitingPlace* is the target *PathNode* of two *NodeConnection*s, the source *PathNode* of exactly one of the two *NodeConnection*s 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 65). 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. For the concrete syntax of the *Timer* subclass of *WaitingPlace*, see clause 8.2.14. See also *Connect* for further visualizations of waiting places.

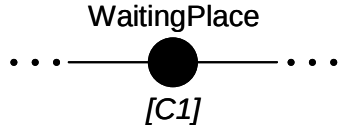


Figure 65 – Symbol: UCM waiting place

Relationships

- Inherits relationships from *PathNode*.

Constraints

e. 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 modeled with a second UCM path that is connected to the waiting place (see *Connect*). A trigger counter keeps track of the arrivals (see *WaitKind* for details, clause 8.2.15). 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*), 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 2 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 timer via the waiting path (i.e., waiting counter > 0). Table 2 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 2 – Overview of waiting place semantics

CWP	*TC = 0*	*TC > 0*
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.

8.2.14 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 54).

a) Abstract grammar

Attributes

- Inherits attributes from *WaitingPlace*.

Relationships

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

Constraints

a. Inherits constraints from *WaitingPlace*.

b. A *Timer* is the source *PathNode* of one or two *NodeConnection*s.

b) Concrete grammar

The symbol for *Timer* on a UCM path is defined as a clock symbol (🕒) with the name of the timer (from superclass *URNmodelElement*) optionally displayed next to the symbol according to *Label* coordinates 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* for further visualizations of timers.

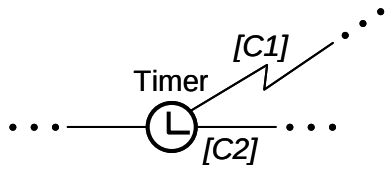


Figure 66 – Symbol: UCM timer with timeout path

Relationships

- Inherits relationships from *WaitingPlace*.

Constraints

c. 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 modeled with a second UCM path that is connected to the waiting place (see *Connect*). A trigger counter keeps track of the arrivals (see *WaitKind* for details, clause 8.2.15). The conditions of the waiting place are the *Condition*s 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*), 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 3 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 3 is read row by row. For example, the last row says that if the CRP evaluates to false (first column) and the CTP 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 3 – Overview of timer semantics

CRP	*CTOP*	*TC = 0*	*TC > 0*
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.

8.2.15 WaitKind

A waiting place can be Transient or Persistent (see Figure 54).

a) Abstract grammar

Attributes

- None (enumeration metaclass).

Relationships

- Used by *WaitingPlace*.

Constraints

None.

b) Concrete grammar

None (enumeration metaclass).

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*).

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*). In this case, the trigger counter is never greater than one, thus modeling that 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*).

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.

8.2.16 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 54).

a) Abstract grammar

Attributes

- Inherits attributes from *PathNode*.

Relationships

- Inherits relationships from *PathNode*.

Constraints

a. Inherits constraints from *PathNode*.

b. A *Connect* is the source *PathNode* of exactly one *NodeConnection*.

c. A *Connect* is the target *PathNode* of exactly one *NodeConnection*.

d. If a *Connect* is the source *PathNode* of a *NodeConnection*, the target *PathNode* of the *NodeConnection*s is of type *WaitingPlace* or *StartPoint*.

e. If a *Connect* is the target *PathNode* of a *NodeConnection*, the source *PathNode* of the *NodeConnection*s is of type *EndPoint* or *EmptyPoint*.

f. 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 67 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.

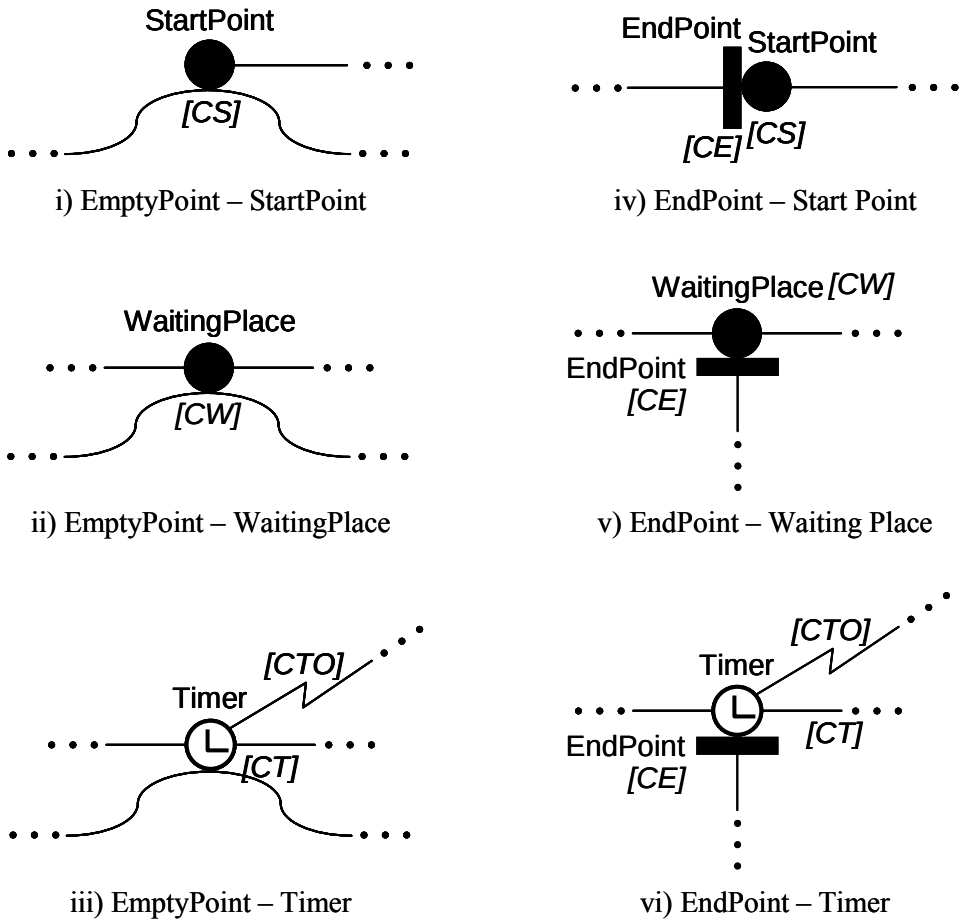


Figure 67 – Examples: UCM connects

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*.

Relationships

- Inherits relationships from *PathNode*.

Constraints

g. Inherits constraints from *PathNode*.

h. A *Connect* does not have a *Label*.

i. A *Connect* does not have a *Position*.

j. There must 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 *NodeConnection*s with connects are summarized in Table 4.

Table 4 – Combinations of node connections with connects

First Node Connection		*Second Node Connection*		*Visualized in*
Source Node	***Target Node***	***Source Node***	***Target Node***	
Empty Point	Connect	Same Connect	Start Point	Figure 67 (i)
Empty Point	Connect	Same Connect	Waiting Place	Figure 67 (ii)
Empty Point	Connect	Same Connect	Timer	Figure 67 (iii)
End Point	Connect	Same Connect	Start Point	Figure 67 (iv)
End Point	Connect	Same Connect	Waiting Place	Figure 67 (v)
End Point	Connect	Same Connect	Timer	Figure 67 (vi)

8.3 UCM stubs and plug-ins

*Stub*s and their *PluginBinding*s 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. *PluginBinding*s specify *ComponentBinding*s (covered in clause 8.4), *InBinding*s, and *OutBinding*s. 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 68).

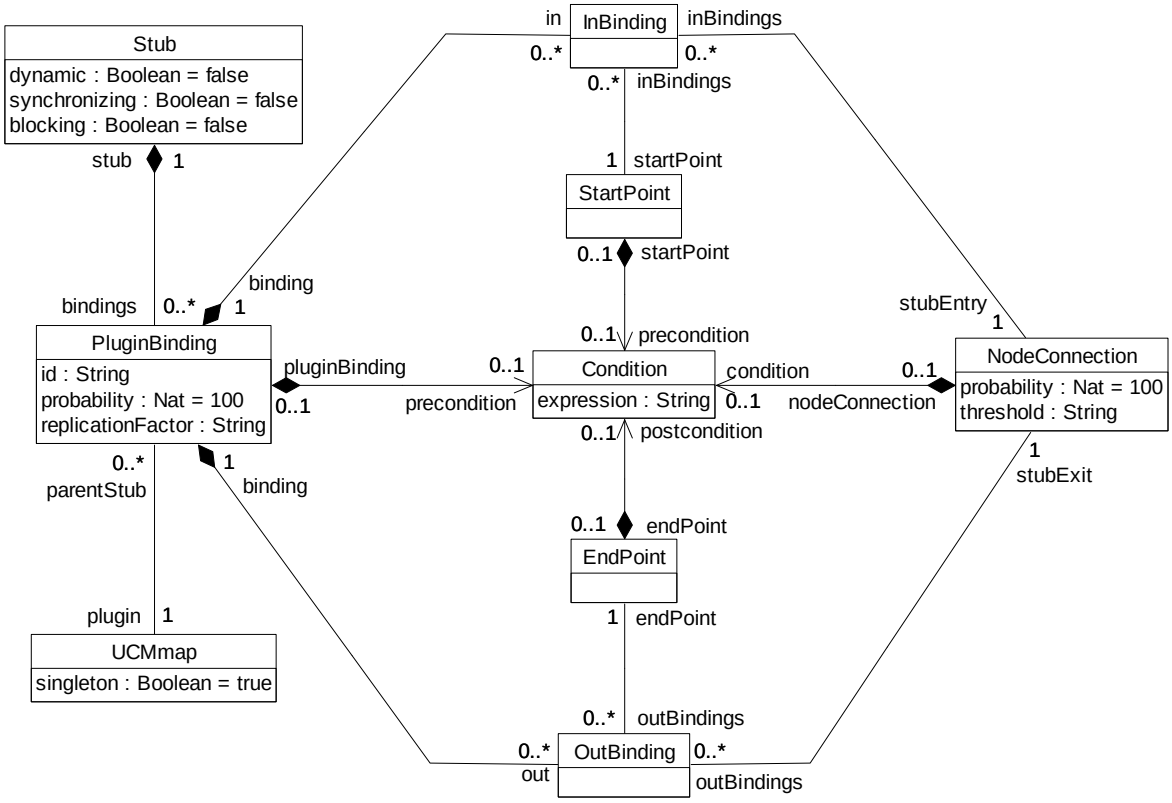


Figure 68 – Abstract grammar: UCM stubs and plug-ins

8.3.1 Stub

Stub is a path node that indicates the presence of hierarchically structured UCM maps (see Figure 54 and Figure 68).

a) Abstract grammar

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.

Relationships

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

Constraints

a. Inherits constraints from *PathNode*.

b. If synchronizing is true, then dynamic is true.

c. If blocking is true, then synchronizing is true.

d. If static is true, then the number of *PluginBinding*s contained by the *Stub* is zero or one.

e. If static is true, then the precondition of *PluginBinding* of the *Stub* is true.

f. 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 (◇) with the name of the stub (from superclass *URNmodelElement*) optionally displayed next to the symbol according to *Label* coordinates of the stub.

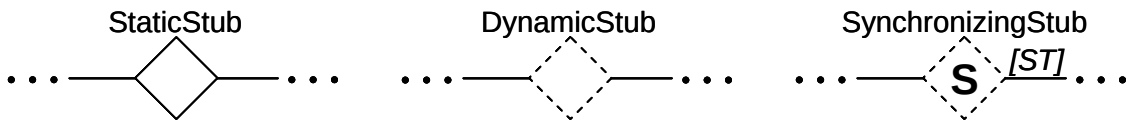


Figure 69 – Symbols: UCM static, dynamic, and synchronizing stubs

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 69). IN and OUT labels may optionally be shown for all kinds of stubs, but the synchronization threshold must be shown for an out-path of a synchronizing stub if the threshold has been defined by the modeller (see Figure 70). 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*.

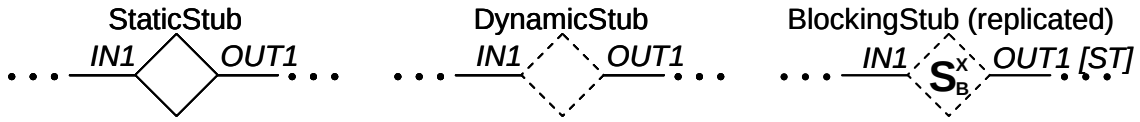


Figure 70 – Symbols: UCM stubs with further annotations

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 71.

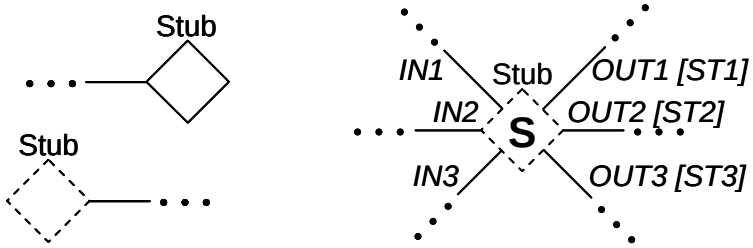


Figure 71 – Example: UCM stubs with different numbers of in-paths and out-paths

Relationships

- Inherits relationships from *PathNode*.

Constraints

g. 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 *PluginBinding*s, *ComponentBinding*s (covered in clause 8.4), *InBinding*s, and *OutBinding*s.

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 *PluginBinding*s 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 *NodeConnection*s 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.

Plug-in map 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 instance must contain not more than one instance of a plug-in map at any time (with the exception of replicated plug-in maps which require that the specified number of instances is created). This also applies to a stub that is used in a loop. The "not more than one map instance per stub 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 72).

- Map G is a singleton and therefore the same 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 instances of Map I.
- Finally, a group of stubs may want to use the same 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 instance of Map I but the group of stubs on Map D, Map E, and Map F uses a different instance of Map I.

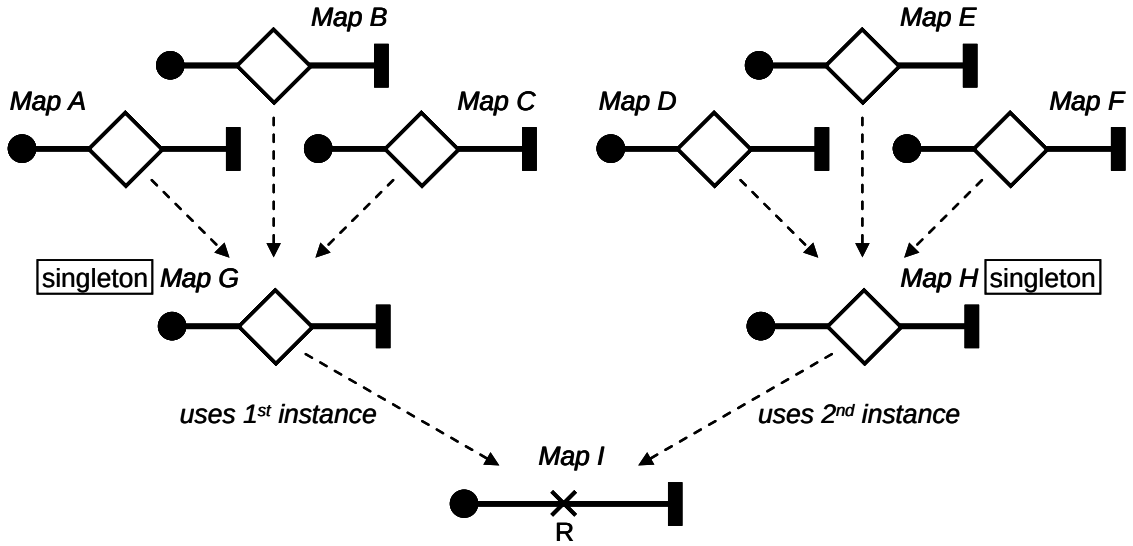


Figure 72 – Example: UCM plug-in map instances

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 *InBinding*s. The out-paths of the stub are merged with the end points on the plug-in map according to the specified *OutBinding*s. The structural specifications on the parent map are merged with the structural specifications on the plug-in map according to the specified *ComponentBinding*s. 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 may contain only one 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 73). Each in-path is equivalent to an AND-fork that is connected to the flattened plug-in maps according to the specified *InBinding*s. Analogously, each out-path corresponds to an OR-join connected to the flattened plug-in maps based on the specified *OutBinding*s. 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.

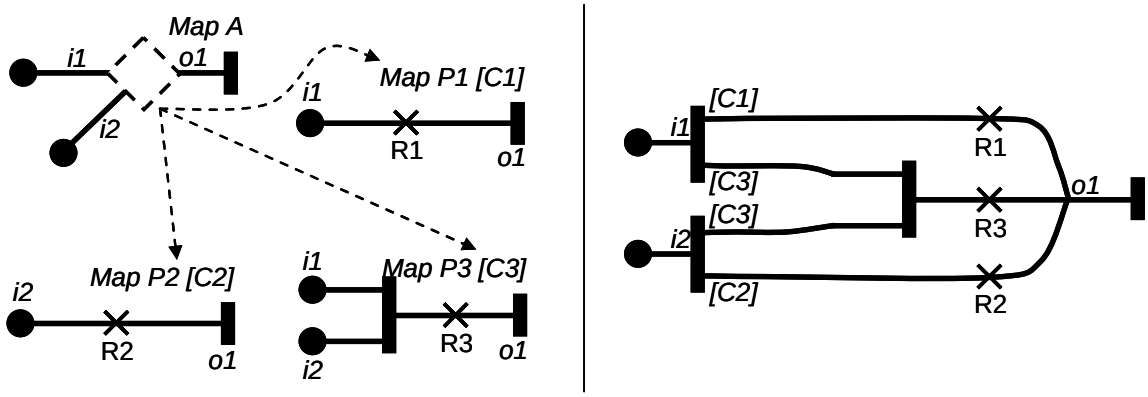


Figure 73 – Example: Semantic flattening of a dynamic stub

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 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 must be created. Therefore, synchronizing stubs can contain more than one instance of a plug-in map at the same time. This behaviour, however, is equivalent to one 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 74) 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 *InBinding*s and *OutBinding*s.

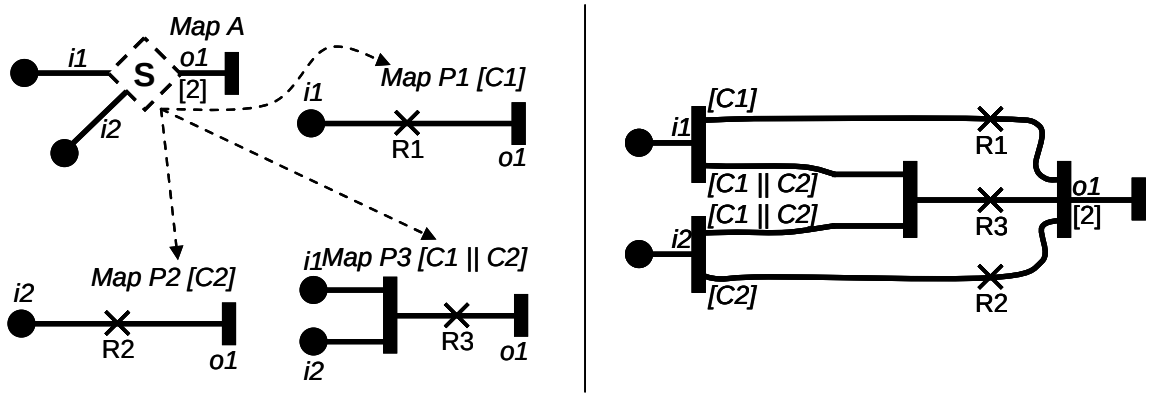


Figure 74 – Example: Semantic flattening of a synchronizing stub

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 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 instances need to be created, because both traversals belong to the same visit. See the UCMs in Figure 75 and the first, second, and third column of Table 5 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 instances of both plug-in maps P1 and P2 are created. The first instances of P1 and P2 are created at the first traversal of in-path i1. The second and third traversals do not cause new instances to be created because these in-paths have not yet been used for the first instances, and hence are part of the same visit. The fourth traversal creates the second set of instances (second visit) because in-path i1 is traversed for the second time. The fifth traversal creates the third set of instances because in-path i1 is again traversed. The sixth and seventh traversals use the instances of the plug-in maps that belong to the second visit because the events go to the longest-waiting instance of a plug-in map. The eighth traversal uses the third set of instances. Finally, the ninth traversal causes the fourth set of 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 instances but not the traversal of start points on the plug-in map (e.g., P1's start point is traversed five times because in-paths i1 and i2 are both bound to the start point). Furthermore, if a replication factor is defined for a plug-in map, then not one instance of the plug-in map is created each time but as many as specified by the replication factor. A replication factor can only be defined for plug-in maps of dynamic stubs.

Table 5 – Instances and synchronizing stubs

#	*In-path*	*Resulting Action {specified synchronization threshold}*	*Resulting Action {default synchronization threshold}*
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 instance) and 1st P2	continue with i2 on 1st P1 (for the second time on that 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 instance) and 2nd P2	continue with i2 on 2nd P1 (for the second time on that 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 75, then the default behaviour stipulates that as many plug-in map instances must arrive at the out-path as are traversed in parallel before the traversal is allowed to continue. The fourth column in Table 5 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 instances that are being traversed changes.

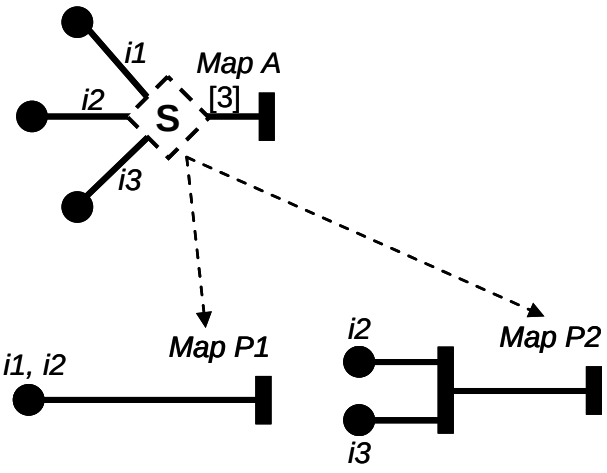


Figure 75 – Example: UCM synchronizing stub with threshold

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 *ComponentBinding*s (covered in clause 8.4), *InBinding*s, and *OutBinding*s. 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 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 Figure 68 and Figure 76).

a) Abstract grammar

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 instances of the plug-in map are used.

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.

Constraints

a. id must be unique within the URN specification.

b. probability ≥ 0 and probability ≤ 100.

c. The replicationFactor must be empty or an Integer expression, as defined in clause 9.3.

d. The replicationFactor must evaluate to a positive Integer value or it may be empty, in which case it is deemed to evaluate to 1.

e. If the singleton attribute of the *UCMmap* of a *PluginBinding* its set to true, then the replicationFactor must evaluate to 1.

f. The *UCMmap* of a *PluginBinding* is the same as the *UCMmap*s that contain the *StartPoint*s that belong to the *InBinding*s of the *PluginBinding*.

g. The *UCMmap* of a *PluginBinding* is the same as the *UCMmap*s that contain the *EndPoint*s that belong to the *OutBinding*s of the *PluginBinding*.

h. The *UCMmap* containing the *Stub* of a *PluginBinding* is the same as the *UCMmap*s that contain the *NodeConnection*s that belong to the *InBinding*s of the *PluginBinding*.

i. The *UCMmap* containing the *Stub* of a *PluginBinding* is the same as the *UCMmap*s that contain the *NodeConnection*s that belong to the *OutBinding*s of the *PluginBinding*.

b) Concrete grammar

None.

c) Semantics

PluginBinding groups together the *ComponentBinding*s, *InBinding*s, and *OutBinding*s 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 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 percent 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 68).

a) Abstract grammar

Attributes

None.

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.

Constraints

a. The target *PathNode* of the *NodeConnection* of an *InBinding* is the *Stub* that contains the *PluginBinding* of the *InBinding*.

b) Concrete grammar

None.

c) Semantics

The traversal of a UCM path utilizes the *InBinding*s 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 68).

a) Abstract grammar

Attributes

None.

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.

Constraints

a. The source *PathNode* of the *NodeConnection* of an *OutBinding* is the *Stub* that contains the *PluginBinding* of the *OutBinding*.

b) Concrete grammar

None.

c) Semantics

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

8.4 UCM components

*Component*s enable modelling of scenario 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 76).

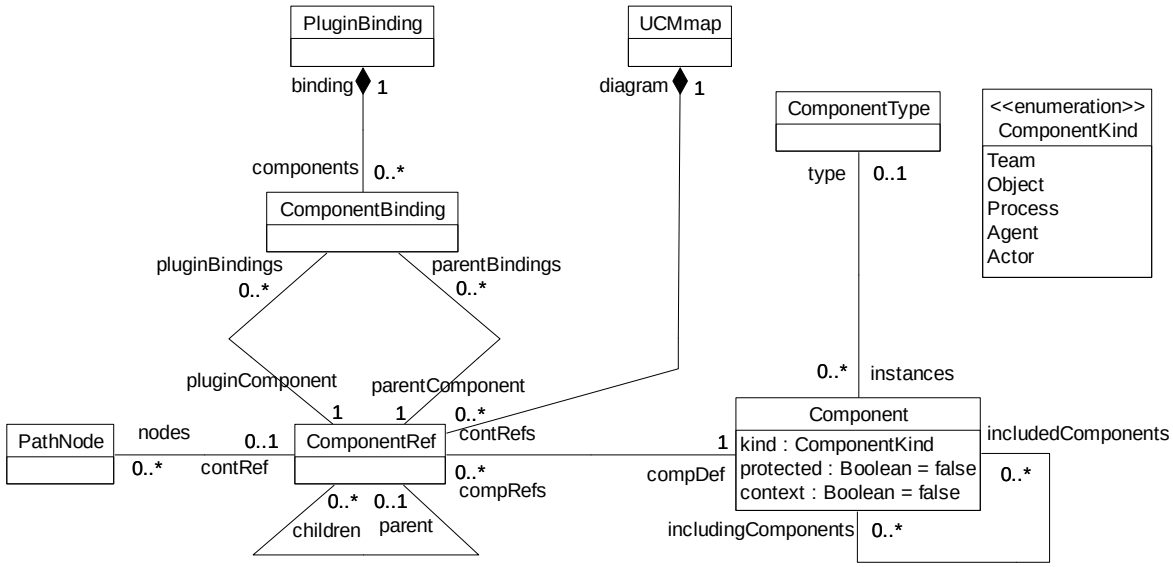


Figure 76 – Abstract grammar: UCM components

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 76).

a) Abstract grammar

Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 53).
- 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.

Relationships

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *Component* definition is contained in the UCM specification (see Figure 52).
- Association with *ComponentType* (0..1): A *Component* definition may have a 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 85).
- Association with *PassiveResource* (0..1): A *Component* definition may correspond to one passive resource (see Figure 85).
- Uses *ComponentKind* enumeration.

Constraints

a. Inherits constraints from *UCMmodelElement*.

b. Any two *Component* definitions cannot share the same name inside a URN specification.

c. The name of a *Component* definition cannot be an empty String.

d. The *Component* containment hierarchy established by the includedComponents relationship does not contain any cycles (i.e., a *Component* definition must not appear more than once on a path from a top node to a leaf node in the containment hierarchy).

b) Concrete grammar

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

Relationships

- Inherits relationships from *UCMmodelElement* (see Figure 53).
- Composition of *ConcreteStyle* (0..1): A *Component* definition may have one concrete style (see Figure 49).

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 *ComponentRef*s 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 subcomponents 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*.

Semantic variations

The following paragraph is non-normative.

Modellers may 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 may be contained at most in only one parent component definition. Furthermore, additional constraints may be imposed in terms of how different kinds of components may be contained in other components. For example, a component of kind Process may not be allowed to be contained in a component of kind Object. Such constraints could also be extended to user-defined *ComponentType*s.

e. A *Component* may be included at the most by one other *Component*.

f. 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 cannot be set to Process.

8.4.2 ComponentType

A *ComponentType* allows the definition of user-defined types of components (see Figure 76).

a) Abstract grammar

Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 53).

Relationships

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *ComponentType* is contained in the UCM specification (see Figure 52).
- Association with *Component* (0..*): A *ComponentType* may be assigned to component definitions.

Constraints

a. Inherits constraints from *UCMmodelElement*.

b) Concrete grammar

A *ComponentType* does not have a visual representation.

Relationships

- Inherits relationships from *UCMmodelElement* (see Figure 53).

c) Semantics

*ComponentType*s 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 76).

a) Abstract grammar

Attributes

- None (enumeration metaclass).

Relationships

- Used by *Component*.

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 76).

a) Abstract grammar

Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 53).

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.

Constraints

a. Inherits constraints from *UCMmodelElement*.

b. The name of a *ComponentRef* must be the same as the name of its associated *Component* definition.

c. The *UCMmap* that contains the *ComponentRef* must be the *UCMmap* that contains *ComponentRef*s associated as children.

d. The containment hierarchy of *ComponentRef*s established by the children relationship does not contain any cycles (i.e., a *ComponentRef* must not appear more than once on a path from a top node to a leaf node in the containment hierarchy).

e. Let CR1 and CR2 be *ComponentRef*s. If CR1 is the parent of CR2, then the *Component* definition of CR1 must 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 77, where the name of each component reference is the same as its kind, for illustration purpose).

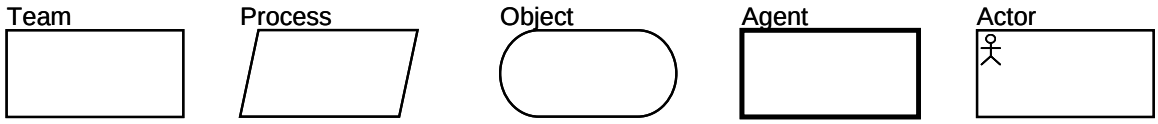


Figure 77 – Symbol: UCM component reference

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 78.

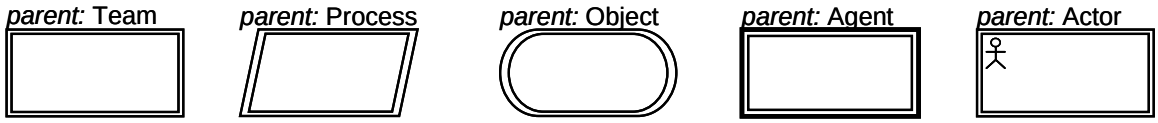


Figure 78 – Symbol: UCM protected and context-dependent component reference

The name of the component reference (from superclass *URNmodelElement*) is displayed next to the symbol according to *Label* coordinates. 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 78). 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 79) for an illustration of these layout principles.

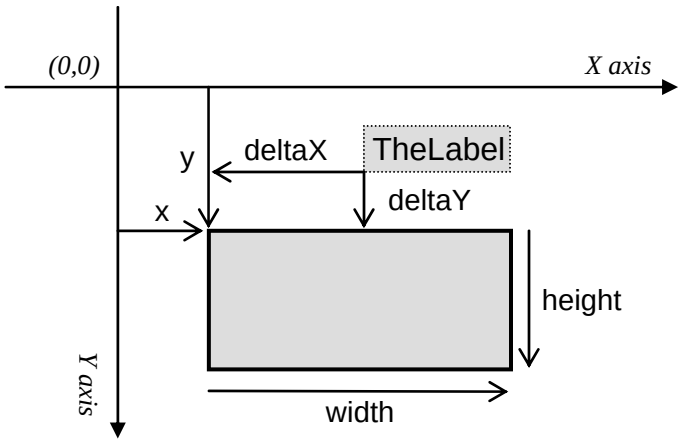


Figure 79 – Layout: Position, size, and label for ComponentRef

Relationships

- Inherits relationships from *UCMmodelElement* (see Figure 53).
- Composition of *Label* (0..1): A *ComponentRef* may have one label (see Figure 46).
- Composition of *Size* (0..1): A *ComponentRef* may have one size (see Figure 48).
- Composition of *Position* (0..1): A *ComponentRef* may have one position (see Figure 47).

Constraints

f. A *ComponentRef* must have one *Label*.

g. A *ComponentRef* must have one *Size*.

h. A *ComponentRef* must have one *Position*.

i. The symbol of a *ComponentRef* must 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 *UCMmap*s. The semantics of *ComponentRef* is defined by the attributes of its *Component* definition and its *ComponentBinding*s (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 must be cycle-free and that C1 must be an ancestor of C3 in the containment hierarchy is not violated (see Figure 80.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 80.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 80.iii).

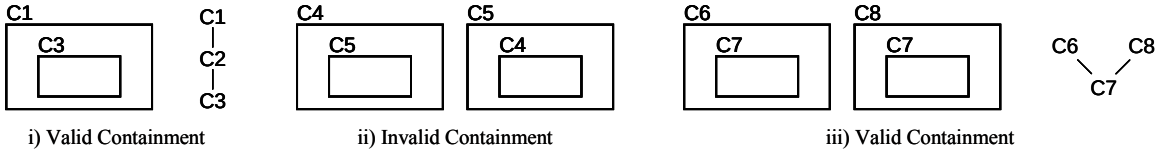


Figure 80 – Examples: UCM component containment hierarchies

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 76).

a) Abstract grammar

Attributes

None.

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.

Constraints

a. The *UCMmap* of the *ComponentRef* associated as parentComponent is the *UCMmap* of the *Stub* of the *PluginBinding* of the *ComponentBinding*.

b. The *UCMmap* of the *ComponentRef* associated as pluginComponent is the *UCMmap* of the *PluginBinding* of the *ComponentBinding*.

c. The context attribute of the *Component* definition of a pluginComponent must be set to true.

d. The kind attribute of the *Component* definition of a pluginComponent must be set to Team.

e. The *Component* definition of a pluginComponent does not have a *ComponentType*.

b) Concrete grammar

None.

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 81. 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).

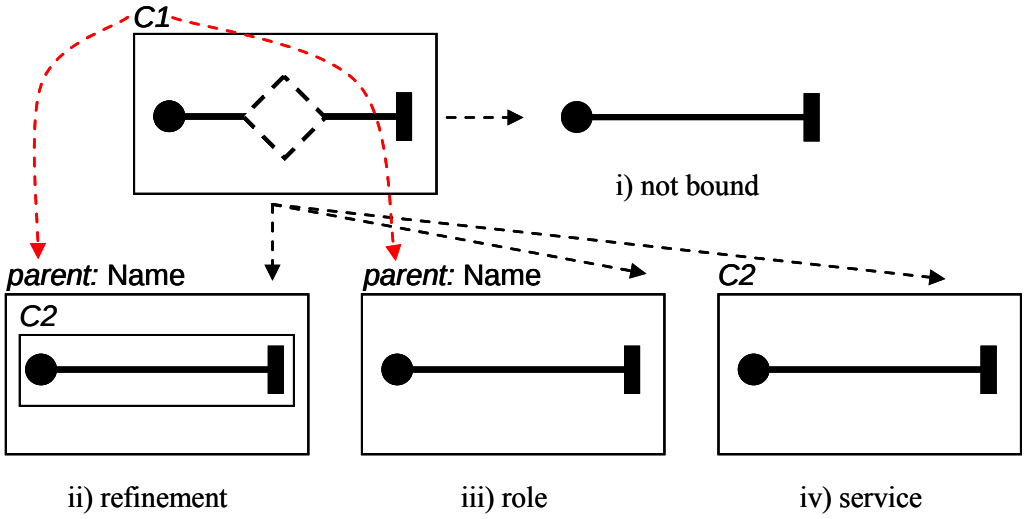


Figure 81 – Example: Plug-in bindings for components

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 modeled as shown in the example in Figure 82. 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.

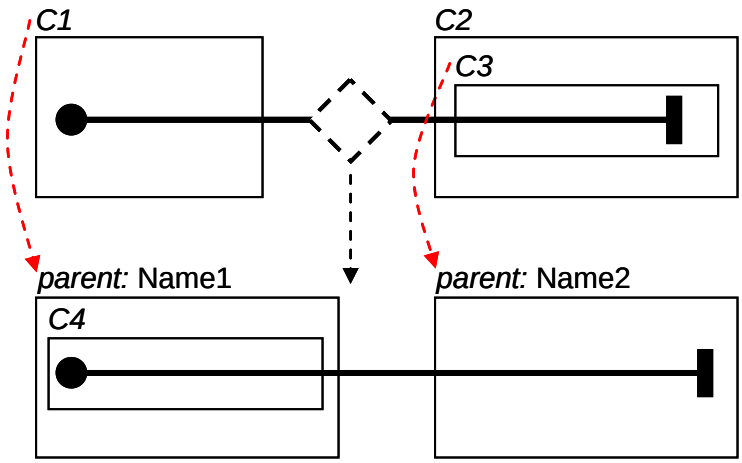


Figure 82 – Example: Plug-in bindings for multiple components

8.5 UCM scenario definitions

Scenario definitions (see *ScenarioDef*) 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 83).

There are no specific concrete grammar metaclasses for the model elements defined in this clause.

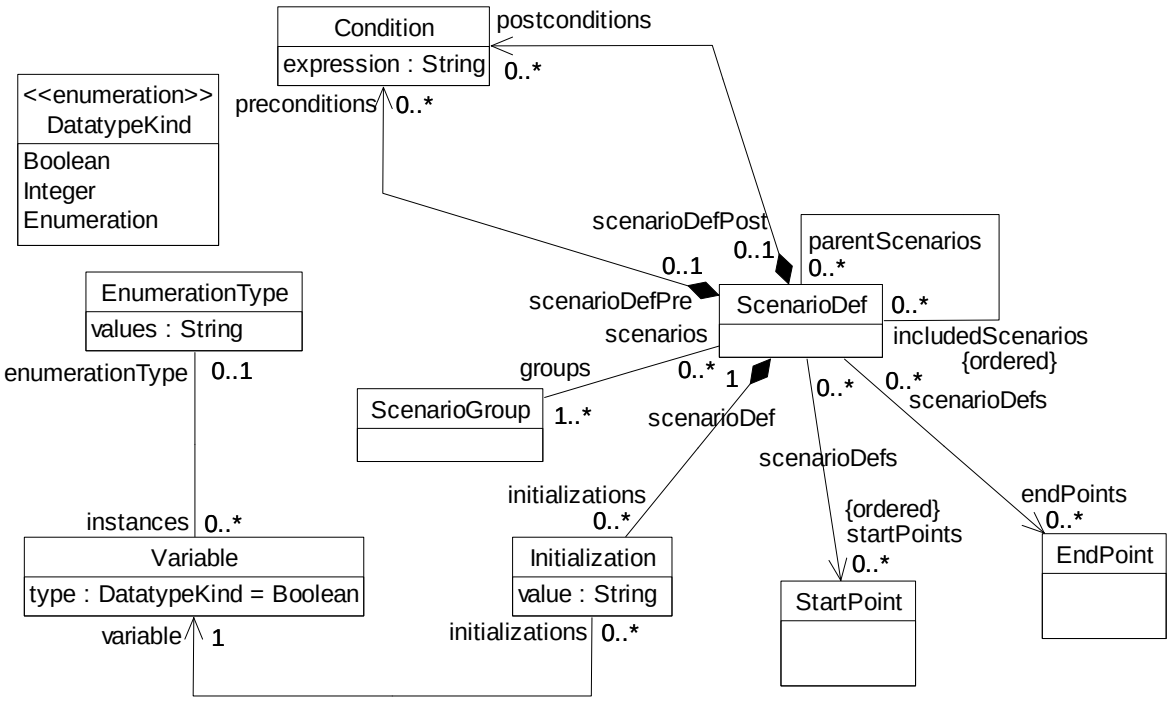


Figure 83 – Abstract grammar: UCM scenario definitions

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 83).

a) Abstract grammar

Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 53).

Relationships

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *ScenarioGroup* is contained in the UCM specification (see Figure 52).
- Association with *ScenarioDef* (0..*): A *ScenarioGroup* may refer to scenario definitions.

Constraints

a. Inherits constraints from *UCMmodelElement*.

b) Concrete grammar

A *ScenarioGroup* does not have a visual representation.

Relationships

- Inherits relationships from *UCMmodelElement* (see Figure 53).

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 83).

a) Abstract grammar

Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 53).

Relationships

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *ScenarioDef* is contained in the UCM specification (see Figure 52).
- 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.

Constraints

a. Inherits constraints from *UCMmodelElement*.

b. The scenario containment hierarchy established by the includedScenarios relationship does not contain any cycles (i.e., a *ScenarioDef* must 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 visual representation.

Relationships

- Inherits relationships from *UCMmodelElement* (see Figure 53).

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*) are satisfied after initializing the variables in the global data model of the URN specification according to the *Initialization*s 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 *EndPoint*s are compared against the desired *EndPoint*s 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 *StartPoint*s, *EndPoint*s, preconditions, and postconditions for its purposes. For conflicting *Initialization*s (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 55. 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 6 are structured according to this approach.

Table 6 – Example: End-to-end UCM scenario definitions

Name	*Start Points*	*Initializations* [1)]	*End Points*
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
[1)] <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 7 show a common scenario definition used by the TL feature.

Table 7 – Example: End-to-end UCM scenario definitions with common elements

Name	***Start Points***	***Initializations*** [1)]	***End Points***
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 ***bold and italic***):			
TL success	request	!subOCS, subTL, !subTCS, !busy, ***!TLactive***, PINvalid	***ring, ringing***
TL pin success	request, ***enterPIN***	!subOCS, subTL, !subTCS, !busy, TLactive, PINvalid	***ring, ringing***
TL busy	request	!subOCS, subTL, !subTCS, ***busy***, ***!TLactive***, PINvalid	***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, PINvalid	***notify***
[1)] <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 8 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 highlighting in **bold**.

Table 8 – Example: Feature-specific UCM scenario definitions

Name [1)]	***Start Points***	***Initializations*** [2)]	***End Points***
Basic Call core	***request***	***!subOCS, !subTL, !subTCS, !busy***	
Basic Call success +Basic Call core	request	!subOCS, !subTL, !subTCS, !busy	***ring, ringing***
Basic Call busy +Basic Call core	request	!subOCS, !subTL, !subTCS, ***busy***	***busy***
OCS core		***subOCS, !onOCSlist***	
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, ***onOCSlist***	***notify***
TL core		***subTL, TLactive, PINvalid***	
TL core PIN +TL core	***enterPIN***	subTL, TLactive, PINvalid	
TL success +Basic Call success +TL core	request	!subOCS, subTL, !subTCS, !busy, ***!TLactive***, 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, ***!TLactive***, 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, ***!PINvalid***	***notify***
TL timeout +Basic Call core +TL core	request	!subOCS, subTL, !subTCS, !busy, TLactive, PINvalid	***notify***
TCS core		***subTCS, !onTCSlist***	
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, ***onTCSlist***	***notify***

Name [1)]	*Start Points*	*Initializations* [2)]	*End Points*
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, ***!PINvalid***, !onOCSlist	***notify***
[1)] + denotes an included scenario [2)] <variable> is shorthand for <variable> = true; !<variable> is shorthand for <variable> = false;			

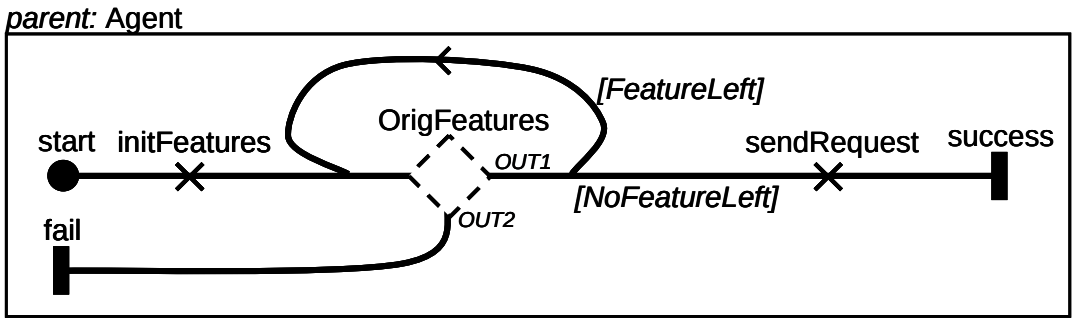
The last two scenario definitions are of particular interest because the more features are combined, the more likely it is that undesired interactions between these features manifest themselves in the UCM model. The second last scenario definition is not problematic, but the last one is. If the UCM path 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 84:

- 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).



i) Originating Features (improved)

Figure 84 – 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 83).

a) Abstract grammar

Attributes

- value (String): The initial value of a variable.

Relationships

- Contained by *ScenarioDef* (1): An *Initialization* is contained in one scenario definition.
- Association with *Variable* (1): An *Initialization* is for one variable.

Constraints

a. The value must be a Boolean literal as defined in clause 9.2.1, if the type of the *Variable* is Boolean.

b. The value must 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.

c. The value must 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

None.

c) Semantics

*Initialization*s 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

*Variable*s 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 83).

a) Abstract grammar

Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 53).
- type (*DatatypeKind*): The type of the variable. Default value is Boolean.

Relationships

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *Variable* is contained in the UCM specification (see Figure 52).
- Association with *EnumerationType* (0..1): A *Variable* may be of Enumeration type.
- Uses *DatatypeKind* enumeration.

Constraints

a. Inherits constraints from *UCMmodelElement*.

b. Any two *Variable*s cannot share the same name inside a URN specification.

c. A *Variable* has one *EnumerationType* if and only if the type of the *Variable* is Enumeration.

d. The name of a *Variable* must be different from the following keywords: and, or, xor, not, mod, true, false, if, else.

e. The name of a *Variable* must not end with "_pre".

b) Concrete grammar

A *Variable* does not have a visual representation.

Relationships

- Inherits relationships from *UCMmodelElement* (see Figure 53).

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 83).

a) Abstract grammar

Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 53).

- values (String): A comma-separated list of values specifies the valid choices for an *Enumeration* type.

Relationships

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): An *EnumerationType* is contained in the UCM specification (see Figure 52).
- Association with *Variable* (0..*): An *EnumerationType* may be used for variables.

Constraints

a. Inherits constraints from *UCMmodelElement*.

b. values must be a comma-separated list of enumeration literals as defined in clause 9.2.3.

c. Each individual value in values must be unique within values.

d. Each individual value in values must be different from all *Variable* names.

e. Each individual value in values must not end with "_pre".

f. Each individual value in values must be different from the following keywords: and, or, xor, not, mod, true, false, if, else.

b) Concrete grammar

An *EnumerationType* does not have a visual representation.

Relationships

- Inherits relationships from *UCMmodelElement* (see Figure 53).

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 83).

a) Abstract grammar

Attributes

- None (enumeration metaclass).

Relationships

- Used by *Variable*.

Constraints

None.

b) Concrete grammar

None (enumeration metaclass).

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 *ExternalOperation*s. 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 85).

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.

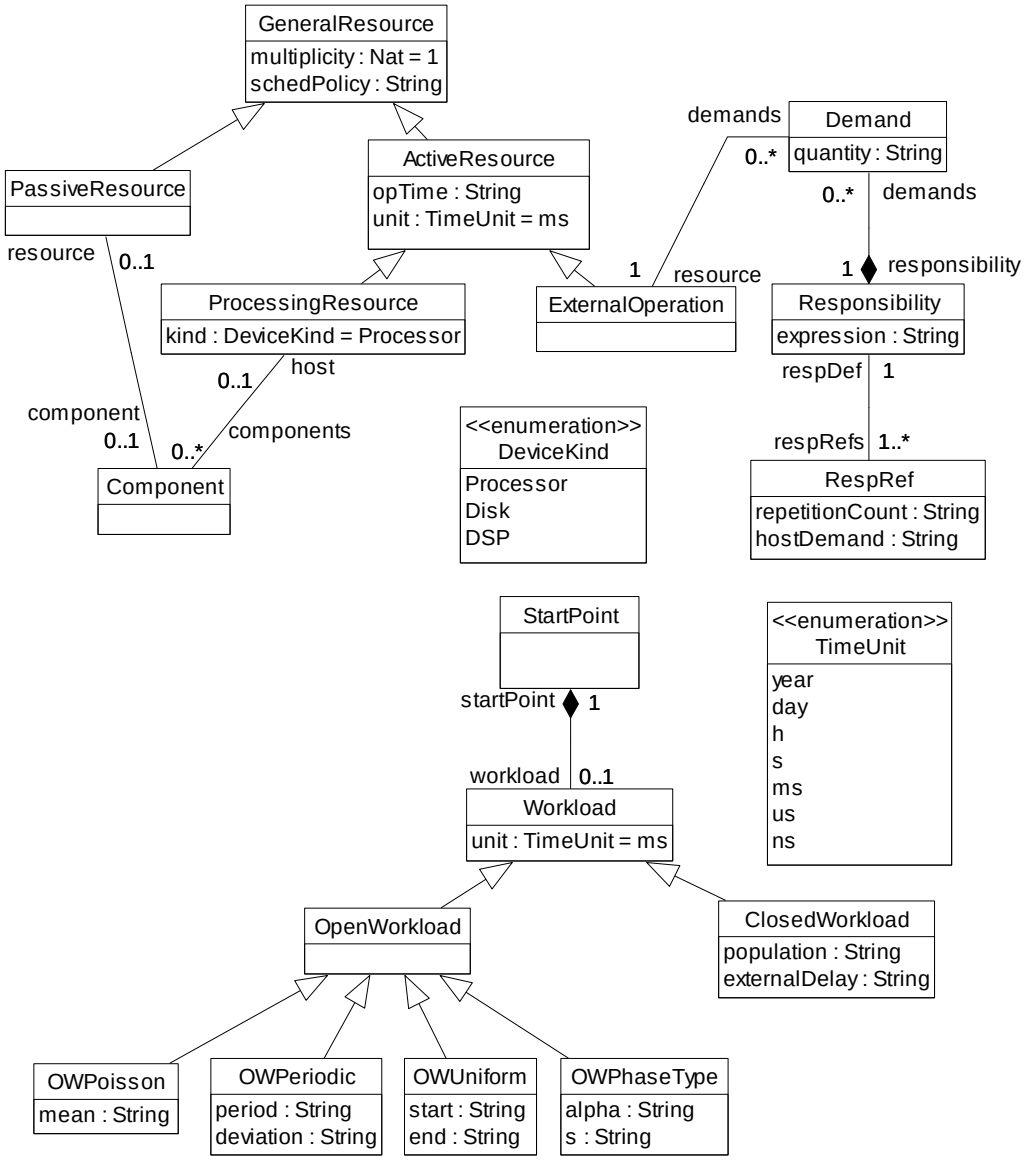


Figure 85 – Abstract grammar: UCM performance annotations

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 85).

a) Abstract grammar

Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 53).
- unit (*TimeUnit*): The unit of time used by the other attributes of the workload. Default value is ms (millisecond).

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*.

Constraints

a. Inherits constraints from *UCMmodelElement*.

b. All instances of *Workload* must appear in one of its subclasses (that is, metaclass *Workload* is abstract).

b) Concrete grammar

A *Workload* does not have a visual representation.

Relationships

- Inherits relationships from *UCMmodelElement* (see Figure 53).

c) Semantics

A *Workload* characterizes the requests for a scenario at a start point. A workload must 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 (µs), or nanoseconds (ns) (see Figure 85).

a) Abstract grammar

Attributes

- None (enumeration metaclass).

Relationships

- Used by *Workload* and *ActiveResource*.

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)
- µs: 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 85).

a) Abstract grammar

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.

Relationships

- Inherits relationships from *Workload*.

Constraints

a. Inherits constraints from *Workload*.

b. The population must be an Integer expression, as defined in clause 9.3.

c. The population must evaluate to a non-negative Integer value.

d. The externalDelay must be an Integer expression, as defined in clause 9.3.

e. The externalDelay must evaluate to a non-negative Integer value.

b) Concrete grammar

A *ClosedWorkload* does not have a visual representation.

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 85).

a) Abstract grammar

Attributes

- Inherits attributes from *Workload*.

Relationships

- Inherits relationships from *Workload*.
- *OpenWorkload* is a superclass of *OWPoisson*, *OWPeriodic*, *OWUniform*, and *OWPhaseType*.

Constraints

a. Inherits constraints from *Workload*.

b. All instances of *OpenWorkload* must appear in one of its subclasses (that is, metaclass *OpenWorkload* is abstract).

b) Concrete grammar

An *OpenWorkload* does not have a visual representation.

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 85).

a) Abstract grammar

Attributes

- Inherits attributes from *OpenWorkload*.
- mean (String): The mean of the Poisson distribution is the value of the mean attribute divided by 1000.

Relationships

- Inherits relationships from *OpenWorkload*.

Constraints

a. Inherits constraints from *OpenWorkload*.

b. The mean must be an Integer expression, as defined in clause 9.3.

c. The mean must evaluate to a positive Integer value.

b) Concrete grammar

An *OWPoisson* does not have a visual representation.

Relationships

- Inherits relationships from *Workload*.

c) Semantics

OWPoisson is an *OpenWorkload* with a mathematical Poisson arrival distribution. The mean of the distribution (often called λ 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 85).

a) Abstract grammar

Attributes

- Inherits attributes from *OpenWorkload*.
- period (String): The period of the periodic arrival.
- deviation (String): The maximal deviation of the periodic arrival.

Relationships

- Inherits relationships from *OpenWorkload*.

Constraints

a. Inherits constraints from *OpenWorkload*.

b. The period must be an Integer expression, as defined in clause 9.3.

c. The period must evaluate to a positive Integer value.

d. The deviation must be an Integer expression, as defined in clause 9.3.

e. The deviation must evaluate to a non-negative Integer value.

b) Concrete grammar

An *OWPeriodic* does not have a visual representation.

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 85).

a) Abstract grammar

Attributes

- Inherits attributes from *OpenWorkload*.
- start (String): The start of the sampling interval.
- end (String): The end of the sampling interval.

Relationships

- Inherits relationships from *OpenWorkload*.

Constraints

a. Inherits constraints from *OpenWorkload*.

b. The start must be an Integer expression, as defined in clause 9.3.

c. The start must evaluate to a non-negative Integer value.

d. The end must be an Integer expression, as defined in clause 9.3.

e. The end must evaluate to a positive Integer value greater than start's evaluated value.

b) Concrete grammar

An *OWUniform* does not have a visual representation.

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 85).

a) Abstract grammar

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.

Relationships

- Inherits relationships from *OpenWorkload*.

Constraints

a. Inherits constraints from *OpenWorkload*.

b) Concrete grammar

An *OWPhaseType* does not have a visual representation.

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 inter-related 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 hypoexponential 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 85).

a) Abstract grammar

Attributes

- Inherits attributes from *UCMmodelElement* (see Figure 53).
- multiplicity (Nat): The number of available resources. Default value is 1.
- schedPolicy (String): The type of scheduling policy.

Relationships

- Inherits relationships from *UCMmodelElement*.
- Contained by *UCMspec* (1): A *GeneralResource* is contained in the UCM specification (see Figure 52).
- *GeneralResource* is a superclass of *PassiveResource* and *ActiveResource*.

Constraints

a. Inherits constraints from *UCMmodelElement*.

b. All instances of *GeneralResource* must appear in one of its subclasses (that is, metaclass *GeneralResource* is abstract).

b) Concrete grammar

A *GeneralResource* does not have a visual representation.

Relationships

- Inherits relationships from *UCMmodelElement* (see Figure 53).

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 85).

a) Abstract grammar

Attributes

- Inherits attributes from *GeneralResource*.

Relationships

- Inherits relationships from *GeneralResource*.
- Association with *Component* (0..1): A *PassiveResource* may have one component definition.

Constraints

a. Inherits constraints from *GeneralResource*.

b) Concrete grammar

A *PassiveResource* does not have a visual representation.

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 must 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 85).

a) Abstract grammar

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).

Relationships

- Inherits relationships from *GeneralResource*.
- *ActiveResource* is a superclass of *ProcessingResource* and *ExternalOperation*.
- Uses *TimeUnit* enumeration.

Constraints

a. Inherits constraints from *GeneralResource*.

b. All instances of *ActiveResource* must appear in one of its subclasses (that is, metaclass *ActiveResource* is abstract).

c. The opTime must be an Integer expression, as defined in clause 9.3.

d. The opTime must evaluate to a non-negative Integer value.

b) Concrete grammar

An *ActiveResource* does not have a visual representation.

Relationships

- Inherits relationships from *GeneralResource*.

c) Semantics

*ActiveResource*s 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 85).

a) Abstract grammar

Attributes

- Inherits attributes from *ActiveResource*.
- kind (*DeviceKind*): The specific kind of hardware processing device represented by the resource. Default value is Processor.

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.

Constraints

a. Inherits constraints from *ActiveResource*.

b) Concrete grammar

A *ProcessingResource* does not have a visual representation.

Relationships

- Inherits relationships from *ActiveResource*.

c) Semantics

A *ProcessingResource* represents a hardware host for the software *Component*s 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 85).

a) Abstract grammar

Attributes

- None (enumeration metaclass).

Relationships

- Used by *ProcessingResource*.

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 85).

a) Abstract grammar

Attributes

- Inherits attributes from *ActiveResource*.

Relationships

- Inherits relationships from *ActiveResource*.
- Association with *Demand* (0..*): An *ExternalOperation* may have demands made on it.

Constraints

a. Inherits constraints from *ActiveResource*.

b) Concrete grammar

An *ExternalOperation* does not have a visual representation.

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. *ExternalOperation*s 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 85).

a) Abstract grammar

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.

Relationships

- Contained by *Responsibility* (1): A *Demand* is contained in one responsibility definition.
- Association with *ExternalOperation* (1): A *Demand* is for one external operation.

Constraints

a. The quantity must be an Integer expression, as defined in clause 9.3.

b. The quantity must 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, 7.6.8, 7.6.10, 7.6.11, 7.6.12, and 7.6.13, respectively (see Figure 6, Figure 46, Figure 47, Figure 48, Figure 49, and Figure 50, respectively). See Figure 97 for a complete overview of the concrete grammar metaclasses for the UCM notation.

Figure 86 – 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 86).

a) Abstract grammar

DirectionArrow has no abstract syntax.

Attributes

- Inherits attributes from *PathNode*.

Relationships

- Inherits relationships from *PathNode*.

Constraints

a. Inherits constraints from *PathNode*.

b) Concrete grammar

The symbol for *DirectionArrow* on a UCM path is defined as an open arrow head (**>**) pointing towards the successor node connection (see Figure 87).

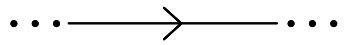


Figure 87 – Symbol: UCM direction arrow

Relationships

- Inherits relationships from *PathNode*.

Constraints

b. Inherits constraints from *PathNode*.

c. A *DirectionArrow* is the source *PathNode* of exactly one *NodeConnection*.

d. 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 *PathNode*s of type *DirectionArrow* as if *DirectionArrow*s do not exist, replacing the two *NodeConnection*s for a *DirectionArrow* (*PathNode* P1 → *DirectionArrow* → *PathNode* P2) with one *NodeConnection* (*PathNode* P1 → *PathNode* P2).

d) Model

None.

e) Example

The example in Figure 88 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.

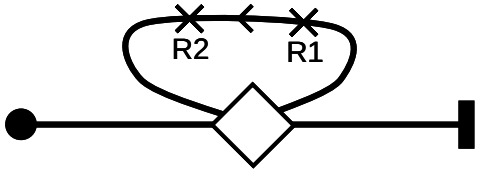


Figure 88 – Example: UCM direction arrow

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 SDLs [ITU-T Z.100]. In order to accommodate modellers less experienced with SDL, 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-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 *Variable*s 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., we cannot assign an Integer value 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 89). These represent a subset of the data types of [ITU-T Z.100], with support for a subset of the SDL syntax, operators, and semantics.

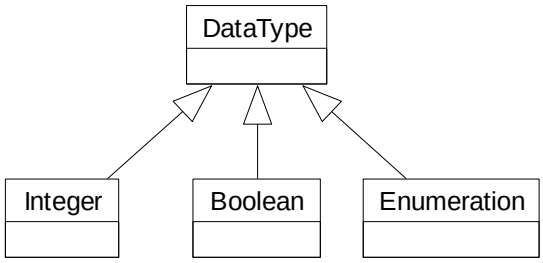


Figure 89 – URN data types

9.2.1 Boolean

The *Boolean* data type corresponds to a subset of the predefined Boolean sort in [ITU-T Z.100]. 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 9 lists the Boolean operators supported in URN, together with their SDL and alternative syntaxes, as well as signatures. The semantics is that of SDL Booleans.

Table 9 – Operators of the Boolean data type

Operator	*SDL syntax*	*Alternative syntax*	*Signature*
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.100]. 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 10 lists the Integer operators supported in URN, together with their SDL and alternative syntaxes, as well as signatures. The semantics is that of SDL Integers. However, unlike SDL's division, URN's division does not generate any exception; attempting to divide an Integer by zero results in an error (e.g., during the traversal of a UCM scenario, see clause 11.2).

Table 10 – Operators of the Integer data type

Operator	*SDL syntax*	*Alternative syntax*	*Signature*
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.100] 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 must not start with a decimal digit and cannot be solely composed of underscores. Each literal must be unique within the enumeration type, and its name must 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). 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>*
                        | <letter> [<word>] <underscore>+
                                   [<word> {<underscore>+ <word>}* <underscore>*]
<word>              ::= {<letter> | <decimal digit>}+
<letter>            ::= (any alphabetical character, with or without accent)
<underscore>        :: =  _
```

Table 11 lists the Enumeration operators supported in URN, together with their SDL 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 must be of the same type (i.e., they are from the same *Enumeration* type).

Table 11 – Operators of the Enumeration data type

Operator	*SDL syntax*	*Alternative syntax*	*Signature*
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 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> | <substraction>}
                                                    <multiplicative expression>}*
<multiplicative expression> ::=
                        <unary expression> {
                           {<multiplication> | <division> | <modulus>}
                           <unary expression> }*
<unary expression> ::=  [<addition> | <substraction>]
                        { <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>          ::=  +
<substraction>      ::=  -
<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>*
```

```
                        | <letter> [<word>] <underscore>+
                                  [<word> {<underscore>+ <word>}* <underscore>*]
<word>              ::= {<letter> | <decimal digit>}+
<letter>            ::= (any printable alphabetical character defined in UCS
                        [ITU-T T.55])
<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 *EnumerationType*s 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> must 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 used-defined enumerations. The different actions, together with their syntax and signatures, are defined in Table 12.

Table 12 – Actions and their signatures

Operator	*SDL syntax*	*Alternative syntax*	*Signature*
Boolean Assignment	:=	=	(Boolean, Boolean) → Void
Integer Assignment	:=	=	(Integer, Integer) → Void
Enumeration Assignment	:=	=	(EnumerationName, EnumerationName) → Void
If Statement	if	if	(Boolean, Void) → Void
If-Else Statement	if else	if else	(Boolean, Void, Void) → Void
Compound Statement	{ }	{ }	(Void)* → 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 12 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> must be the same, otherwise an error is generated. One cannot assign a value to a *Variable* whose name ends with "_pre" (see clause 8.5.4), as such 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.

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 defined in [ITU-T T.55].

This schema is a serialisation 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 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.

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 must 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 GRL *IntentionalElement* and each GRL *Actor*, two new *evaluation values* shall be created: one that is qualitative (qualitativeVal of type *QualitativeLabel*) and the one that is quantitative (quantitativeVal, an *Integer* value in the range [-100..100]).

A GRL model is evaluated by assigning satisfaction values to a subset of the intentional elements (this is done with an *EvaluationStrategy*) and then propagating these values to the other intentional

elements via the *ElementLink*s connecting them. Each *Evaluation* of an *EvaluationStrategy* shall see its values copied to its associated *IntentionalElement*: evaluation is copied to quantitativeVal and qualitativeEvaluation is copied to qualitativeVal. The quantitativeVal and qualitativeVal evaluation attributes of all other intentional elements and of all actors shall be initially set to 0 and None 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 *IntentionalElement*s, and the new quantitativeVal attribute of intentional elements initialized from the selected *EvaluationStrategy*.
- *Qualitative evaluation*: uses the qualitative contribution attribute of *Contribution*, the importance attribute of *IntentionalElement*s, and the new attribute qualitativeVal of intentional elements initialized from the selected *EvaluationStrategy*.
- *Hybrid evaluation*: uses another combination of the above three categories of attributes (for contribution, importance, and intentional 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 *IntentionalElement*s 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 (targets) of the model. The results of selected alternatives can be analysed and conflicts detected.
- *Backward propagation*: the *EvaluationStrategy* initializes some of the *IntentionalElement*s 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 (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 *IntentionalElement*s that are neither leaves nor roots. From these elements, forward propagation is used to compute evaluation values of higher-level intentional elements whereas backward propagation is used to compute evaluation values of lower-level intentional elements.

c) **Actor satisfaction**: An algorithm may evaluate actor evaluation levels, or not.

d) **Automation**: An algorithm may be fully automated, or interactive (e.g., to resolve conflicts).

e) **Cycles**: An algorithm may handle cycles in GRL models (completely or partially), or require models to be acyclic.

f) **Conflicts**: An algorithm may determine that multiple contributions targeting the same intentional elements are conflicting, or not. In addition, if conflicts are detected, then there could be one or many categories of conflicts.

g) **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 that are also initialized by the strategy, but with different values.

h) **Evaluation overriding**: An algorithm may allow some evaluations defined as part of a strategy to be overridden during the propagation, or not.

i) **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.

j) **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.

k) **Link evaluations**: An evaluation algorithm should provide functions to compute results of decomposition, contribution, and dependency link usages.

l) **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 out of scope for 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 *IntentionalElementRef*s in GRL diagrams should be updated to reflect the current evaluation values (quantitativeVal and qualitativeVal) of the referenced intentional element (see Figure 32 and Figure 33). Similarly, the presentation of *ActorRef* and *CollapsedActorRef* in GRL diagrams should be updated to reflect the current evaluation values (quantitativeVal and qualitativeVal) of the referenced actors (see Figure 27, Figure 29, and Figure 35).

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 must be merged with the corresponding elements from the scenario definition itself as defined in the semantics clause of 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.

The current requirements for the path traversal mechanism (Table 13) 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*, and *visit* which are shown in **bold and underline** in Table 13 and are defined in clause 3.

Table 13 – Requirements for path traversal mechanism

ID	*Requirement*
1	*Path Traversal* (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_1, B_2, … and B_N if a) the traversal is currently visiting path element A, and b) there is a direct node connection from A to B_1, from A to B_2, …, and from A to B_N, and c) the *Path Continuation Criteria* (PCC) of A is fulfilled.
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 d) it cannot move to another path element from any of the currently visited path elements and e) there is no unused start point remaining in the scenario definition.

ID	*Requirement*
10	PT shall issue a warning if a) the traversal has stopped and b) the traversal is currently visiting at least one path element other than an end point.
11	PT shall issue an error if the traversal has stopped and a) at least one end point of the scenario definition has not been visited or b) at least one postcondition of the scenario definition is not fulfilled or c) a postcondition of at least one currently visited end point is not fulfilled.
12	The PCC for a 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.
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 forcible 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, PT shall increase the number of arrived waiting paths by 1 (the initial number of arrived waiting paths is 0).
30	Upon arrival at a persistent waiting place or persistent timer along the trigger/release path, PT shall increase the number of arrived trigger/release paths by 1 (the initial number of arrived trigger/release paths is 0).
31	Upon arrival at a transient waiting place or transient timer along the trigger/release path, PT shall set the number of arrived trigger/release paths to 1, if the number of arrived waiting paths is greater than 0 (the initial number of arrived trigger/release paths is 0).

ID	*Requirement*
32	The PCC for a waiting place W shall be fulfilled if a) W's condition evaluates to true or b) at least one waiting path and at least one trigger path has arrived at W.
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 of a timer T shall be fulfilled if the condition of T's regular path evaluates to true.
35	The PCC of the regular path of a timer T shall be fulfilled if a) the condition of T's regular path evaluates to false, and b) the condition of T's timeout path evaluates to false, and c) at least one waiting path and at least one trigger path have arrived at T.
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 the number of arrived waiting paths by 1 when continuing past the waiting place or timer unless the number of arrived waiting paths is already 0.
38	PT shall decrease the number of arrived trigger/release paths by 1 when continuing past the waiting place or timer unless the number of arrived trigger/release paths is already 0.
39	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.
40	Upon arrival at a stub S, PT shall first traverse in parallel the plug-in maps of S before continuing with the traversal.
41	Upon arrival at a dynamic stub S, PT shall traverse in parallel the number of instances of a plug-in map M of S as specified by the replication factor of the plug-in binding of M.
42	For each plug-in map instance M of stub S, PT shall move in parallel from S to start points SP_1, SP_2, … and SP_N of plug-in map instance M if a) the traversal is currently visiting path element S, and b) the traversal has reached the stub via a direct node connection N from path element A to S, and c) there is an in-binding of M from N to SP_1, from N to SP_2, …, and from N to SP_N, and d) the PCC of the plug-in map instance M is fulfilled.
43	The PCC for the plug-in map instance of a static stub shall be always fulfilled.
44	The PCC for a plug-in map instance M of a dynamic stub shall be fulfilled if the precondition of the plug-in binding for M evaluates to true.
45	The PCC for an out-path N from the non-synchronizing stub S to a path element shall be fulfilled if a) the traversal is visiting an end point E on a plug-in map instance M of S, and b) an out-binding from E to N exists for M.
46	The PCC for an out-path N from the synchronizing stub S to path element B shall be fulfilled if a) the traversal has visited end points E_1, E_2, … or E_N on a plug-in map instance M of S as often as specified by N's synchronization threshold during the same **visit**, and b) out-bindings from E_1, E_2, … and E_N to N exist.
47	PT shall synchronize a synchronizing stub's plug-in map instances, only if they belong to the same **visit**.

ID	*Requirement*
48	Once for each **visit** upon first arrival at a synchronizing stub S with the default synchronization threshold for an out-path, PT shall set the synchronization threshold of S's out-path to the number of S's plug-in map instances with preconditions evaluating to true.
49	PT shall ignore the arrival of plug-in map instances at an out-path of a synchronizing stub S during a **visit** if the synchronization threshold of the S's out-path has been reached for the **visit**.
50	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 instances of S have been traversed.
51	When all plug-in map instances of a synchronizing stub S have been traversed in the N^{th} **visit**, PT shall treat an in-path of S as having been visited N times, if the in-path was visited less than N times.
52	Upon arrival at a singleton map M, PT shall traverse the only instance of M that exists in the UCM model.
53	Upon arrival at an **unconnected start point** S of a non-singleton **traversal root map** M, PT shall traverse the N^{th} instance of M where N is the number of times S has been visited in the current traversal.
54	Upon arrival at a non-singleton plug-in map M of a non-synchronizing stub S, PT shall traverse a) a different instance of M per different instance of a stub and b) the same instance of M for the same instance of S.
55	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 instances of M per different instance of a stub and b) the same set of replicated instances of M for the same instance of S.
56	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 instance of M per different instance of a stub, and b) the N^{th} instance of M for this instance of S during the N^{th} **visit**, and c) the same instance of M for the same instance of S in the same **visit**.
57	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 instances of M per different instance of a stub, and b) the N^{th} set of replicated instances of M for this instance of S during the N^{th} **visit**, and c) the same set of replicated instances of M for the same instance of S in the same **visit**.
58	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 C.
59	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.
60	Upon arrival at a component reference C on a plug-in map instance with a component plug-in binding to a component reference CP on the parent map instance, PT shall use the component definition of CP as the component definition of C.
61	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 14 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 14 lists all these attributes.

For each requirement in Table 14, 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 14 – Compliance statement of this Recommendation against [ITU-T Z.150] language requirements

ID	*Z.150 Language Requirement*	*Type*	*E/D*	*Z.151 Clauses*	*Explanations*
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 satisfiability 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 tradeoffs 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, organisational, 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

ID	*Z.150 Language Requirement*	*Type*	*E/D*	*Z.151 Clauses*	*Explanations*
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
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	7.6.2, 8.2	GRL graphs and UCM maps
26	Support the reuse of goals, NFRs, and knowledge in general.	NFR	D		Tool issue
27	Support the mapping of input events and preconditions to output events and postconditions in various degrees of detail.	FR	E	8.2	UCM maps
28	Specify the set of input events at a scenario start point.	FR	E	8.2.6	Start point
29	Specify the set of output events at a scenario end point.	FR	E	8.2.7	End point
30	Specify preconditions at scenario start points.	FR	E	6.1.6, 8.2.6	Preconditions of start points
31	Specify postconditions at scenario end points.	FR	E	6.1.6, 8.2.7	Post-conditions of end points
32	Specify input sources (human or machine).	FR	E	8.4	Components
33	Specify output sinks (human or machine).	FR	E	8.4	Components
34	Specify responsibilities and references to these responsibilities.	FR	E	8.2.4, 8.2.5	Responsibilities and responsibility references
35	Specify system operations as causal flows of responsibilities (paths).	FR	E	8.2.2, 8.2.3	Path nodes and node connections
36	Specify alternative paths.	FR	E	8.2.8	Or-forks
37	Specify common paths.	FR	E	8.2.9	Or-joins
38	Specify condition-based decision-making at branching points.	FR	E	6.1.6, 8.2.8	Conditions on Or-forks
39	Define a data model and expression evaluator to express and evaluate conditions at branching points.	FR	E	9, 6.1.6, 8.2.8	URN data language, with use in conditions on Or-forks
40	Specify parallel or concurrent paths.	FR	E	8.2.10	And-forks
41	Specify synchronization of paths within a scenario.	FR	E	8.2.11	And-joins
42	Specify synchronization between paths from multiple scenarios.	FR	E	8.2.13, 8.2.15, 8.2.16	Waiting places, wait kinds, and connects
43	Specify timed synchronization, with a timeout path.	FR	E	8.2.14, 8.2.15, 8.2.16	Timers, wait kinds, and connects
44	Specify repetitive actions within a scenario.	FR	E	8.2.5	Responsibility reference with repetitionCount attribute

ID	*Z.150 Language Requirement*	*Type*	*E/D*	*Z.151 Clauses*	*Explanations*
45	Support hierarchical decomposition of scenarios.	FR	E	8.3	Stubs and plug-ins
46	Specify sub-scenarios as scenarios.	FR	E	8.3	Stubs and plug-ins
47	Specify sub-scenario preconditions and postconditions.	FR	E	8.3.2, 6.1.6	Conditions on plug-in bindings
48	Specify scenario containers with multiple sub-scenarios.	FR	E	8.3.1	Dynamic stubs
49	Define a data model and expression evaluator to select sub-scenarios in dynamic containers.	FR	E	9, 6.1.6, 8.3.1	URN data language, with use in conditions on dynamic stubs
50	Group related scenarios.	FR	E	8.2, 8.5.1	UCM maps and scenario groups
51	Extract individual scenarios from grouped scenarios.	FR	E	8.5.2, 11.2	Scenario definitions and UCM path traversal mechanism
52	Specify individual scenarios using a data model and initializations.	FR	E	8.5.3, 9	Initializations and URN data model
53	Express desirable feature interactions in scenarios.	FR	E	8.5.2, 11.2	Scenario definitions and successful UCM path traversal mechanism
54	Detect undesirable feature interactions in scenarios.	FR	E	11.2	UCM path traversal mechanism errors and warnings
55	Specify components and references to these components.	FR	E	8.4.1, 8.4.4	Components and component references
56	Specify scenarios without reference to components.	FR	E	8.2.2	Path nodes do not need to be bound to components
57	Specify scenarios where scenario elements are allocated to components.	FR	E	8.2.2, 8.4.4	Path nodes may be bound to components
58	Specify abstract components and COTS	FR	E	8.4.2, 8.4.5	Component types and component bindings
59	Specify dynamic entities.	FR	E	8.4.3, 8.4.5	Component kinds and component bindings
60	Specify system boundaries.	FR	E	8.4.1, 8.4.3	Components of kind other than Actor
61	Specify the behaviour of the system's environment.	FR	E	8.2.2, 8.4.4	Path nodes not bound to components
62	Specify actors external to the system.	FR	E	8.4.1, 8.4.3	Components of kind Actor
63	Support backward traceability from URN to source documents.	URN	E	6.1.2, 6.1.4	Unique model element identifier, metadata
64	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
65	Support facilities to connect URN elements to external requirements objects.	URN	E	6.1.2, 6.1.4	Unique model element identifier, metadata
66	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
67	Support traceability between operational aspects of goal/NFR models and responsibilities/scenarios in scenario models.	URN	E	6.1.3	URN links

ID	*Z.150 Language Requirement*	*Type*	*E/D*	*Z.151 Clauses*	*Explanations*
68	Support traceability between performance constraints in NFR models and responsibilities/scenarios/response-time requirements in scenario models.	URN	E	6.1.3	URN links
69	Support the testing of requirements.	URN	E	8.5	Scenario definitions
70	Support testing based on requirements.	FR	E	8.5	Scenario definitions
71	Enable preliminary analysis of performance properties.	URN	E	8.6	UCM performance annotations
72	Attach performance/workload annotations to scenario elements.	FR	E	8.6	UCM performance annotations
73	Specify the environment's processing capacity, network delays, and services provided.	FR	E	8.6	UCM performance annotations
74	Specify response times in terms of target fragments of scenarios.	FR	E	8.5.2, 8.6	Scenario definitions, UCM performance annotations
75	Specify identifiers for the model elements.	URN	E	6.1.2	URNmodelElement with unique id attribute
76	Specify document versions.	URN	E	6.2.1	specVersion
77	Support a graphical representation of requirements.	URN	E	6.2, 7.6, 8.7	Concrete grammar metaclasses and graphical concrete syntax
78	Support a tool-oriented interchange format.	URN	E	10	URN interchange format
79	Support textual annotations traceable to graphical elements.	URN	E	7.6.13	Comments
80	Support textual annotations displayable on conventional media.	URN	E	7.6.13	Comments

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: 20080903
-->
<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: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>
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!--  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: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 minOccurs="0" name="style" type="ConcreteStyle"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="elems" type="xsd:IDREF"/>
                <!-- IntentionalElement -->
      </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="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>
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!--  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>
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!--  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="intElement" type="xsd:IDREF"/>  <!-- IntentionalElement -->
  </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: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>
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!--  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:complexType>
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!--  GRLLinkableElement  -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<xsd:complexType name="GRLLinkableElement">
  <xsd:complexContent>
    <xsd:extension base="GRLmodelElement">
      <xsd:sequence>
        <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: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="IntentionalElement"/>
    <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:element minOccurs="0" name="info" type="ConcreteGRLspec"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!--  InBinding  -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<xsd:complexType name="InBinding">
  <xsd:sequence>
    <xsd:element name="startPoint" type="xsd:IDREF"/>  <!-- StartPoint -->
    <xsd:element name="stubEntry" type="xsd:IDREF"/>  <!-- NodeConnection -->
  </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="GRLLinkableElement">
      <xsd:sequence>
        <xsd:element name="type" type="IntentionalElementType"/>
        <xsd:element default="AND" name="decompositionType" type="DecompositionType"/>
        <xsd:element default="None" name="importance" type="ImportanceType"/>
```

```
      <xsd:element name="importanceQuantitative" type="xsd:integer"/>
      <xsd:element maxOccurs="unbounded" minOccurs="0" name="refs" type="xsd:IDREF"/>
            <!-- IntentionalElementRef -->
      <xsd:element minOccurs="0" name="style" type="ConcreteStyle"/>
      <xsd:element minOccurs="0" name="actor" type="xsd:IDREF"/> <!-- Actor -->
    </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"/> <!-- IntentionalElement -->
    </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>
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!--  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="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:string"/>
    <xsd:element default="100" name="probability" type="xsd:nonNegativeInteger"/>
    <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>
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!--  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" 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 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>
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!--  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="xsd:IDREF"/> <!-- 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>
```

Appendix I

Summary of the URN Notation

(This appendix does not form an integral part of this Recommendation)

I.1 Summary of Abstract Metamodel

Figure 90 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. *Concern*s are shown because they encapsulate also both GRL and UCM model elements.

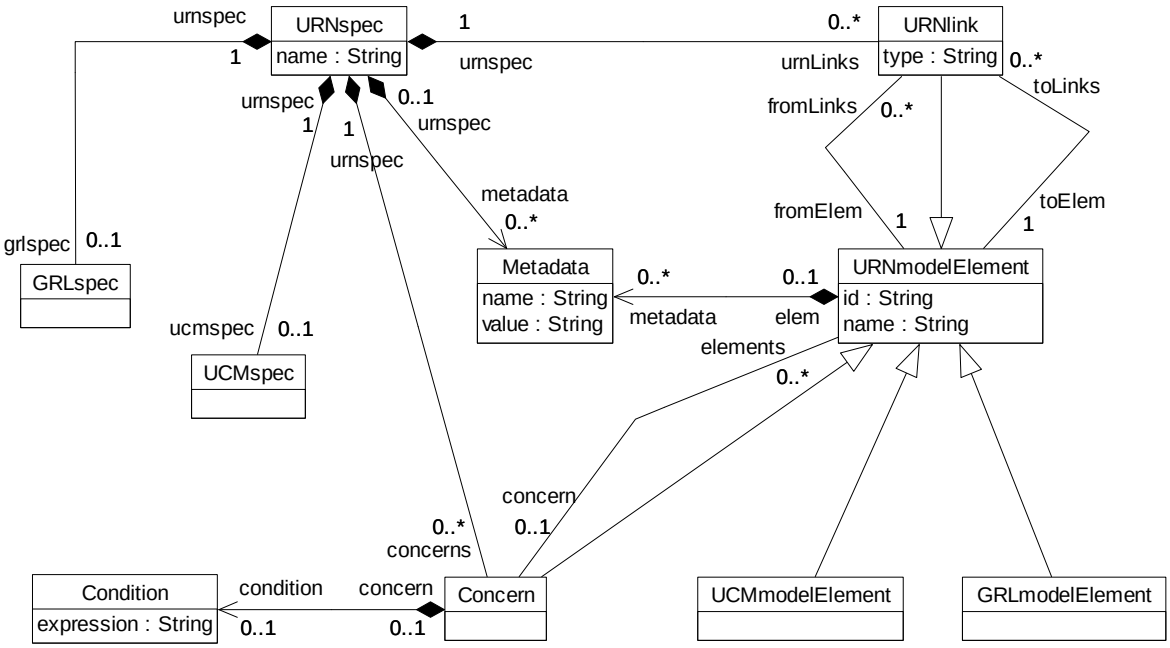


Figure 90 – Abstract grammar: URN top level

Figure 91 presents the complete abstract metamodel of GRL. The diagram is linked to Figure 90 via *GRLspec* and *GRLmodelElement*. The diagram shows a) the relationship of *GRLLinkableElement*s with *ElementLink*s on the left hand side and in the middle and b) the concepts required for the evaluation of GRL models (*StrategiesGroup*, *EvaluationStrategy*, and *Evaluation*), on the right hand side.

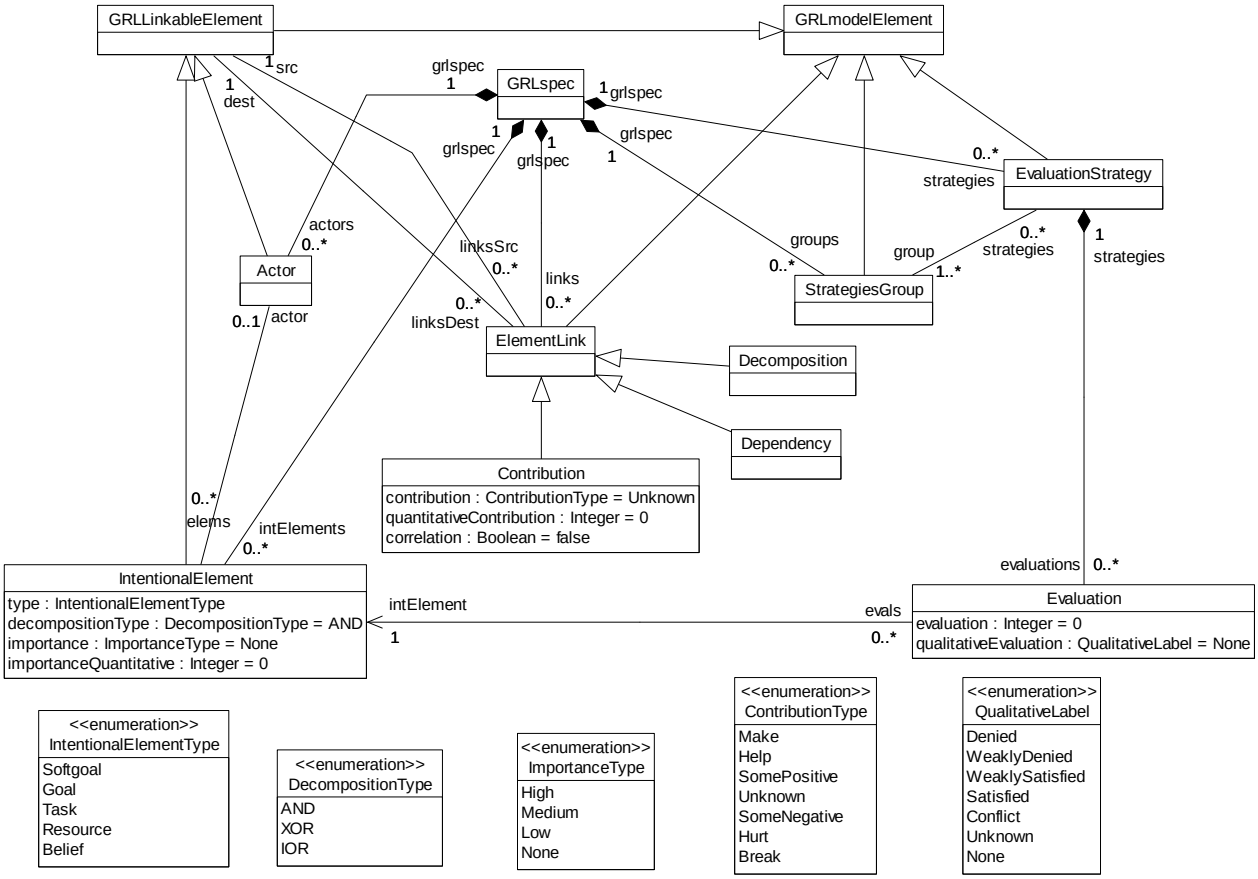


Figure 91 – Abstract grammar: GRL

Figure 92 presents the core abstract metamodel of the UCM notation. The diagram is linked to Figure 90 via *UCMspec* and *UCMmodelElement*. The diagram roughly shows path-related concepts at the top, plug-in binding-related concepts on the left middle, and component-related concepts at the bottom of the figure.

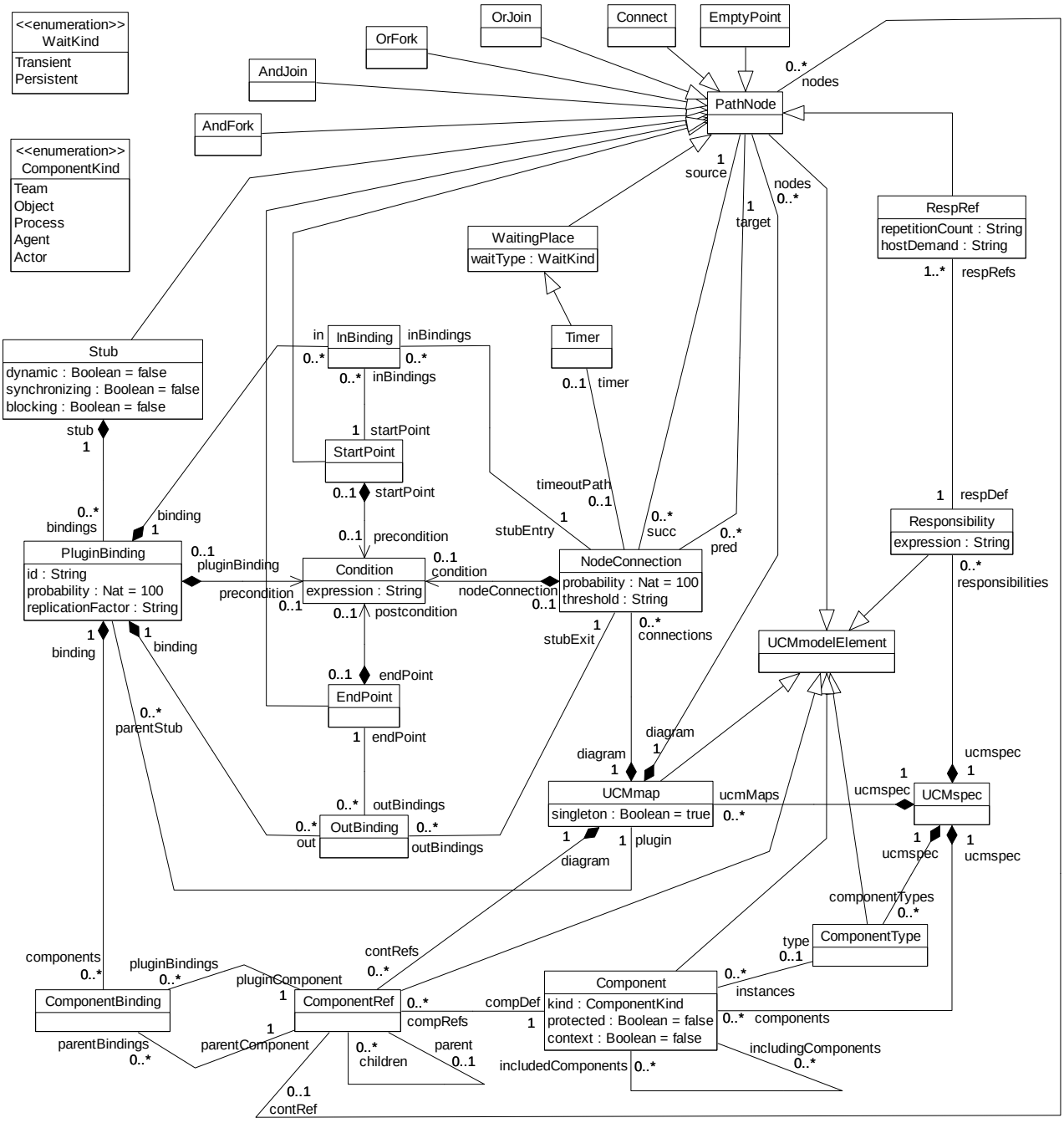


Figure 92 – Abstract grammar: UCM core overview

Figure 93 presents the abstract metamodel of UCM scenarios. The diagram is linked to Figure 90 via *UCMspec* and *UCMmodelElement*. The diagram also introduces further relationships for *StartPoint* and *EndPoint*.

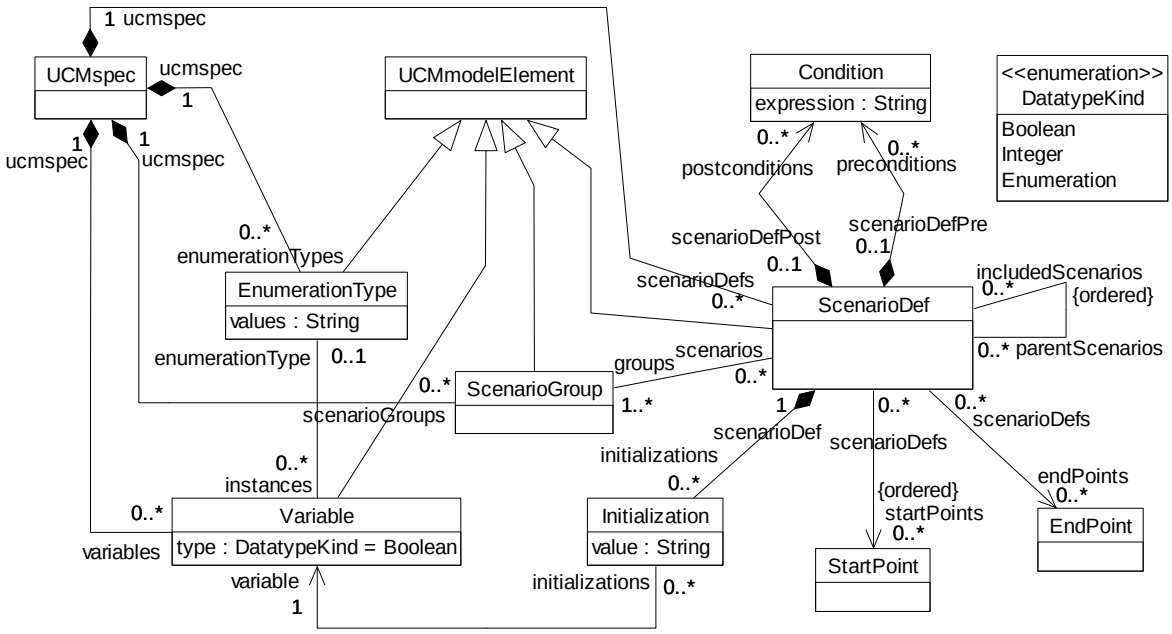


Figure 93 – Abstract grammar: UCM scenarios overview

Figure 94 presents the abstract metamodel of the performance annotations for the UCM notation. The diagram is linked to Figure 90 via *UCMspec* and *UCMmodelElement*. The diagram also introduces further relationships for *StartPoint*, *Responsibility*, and *Component*.

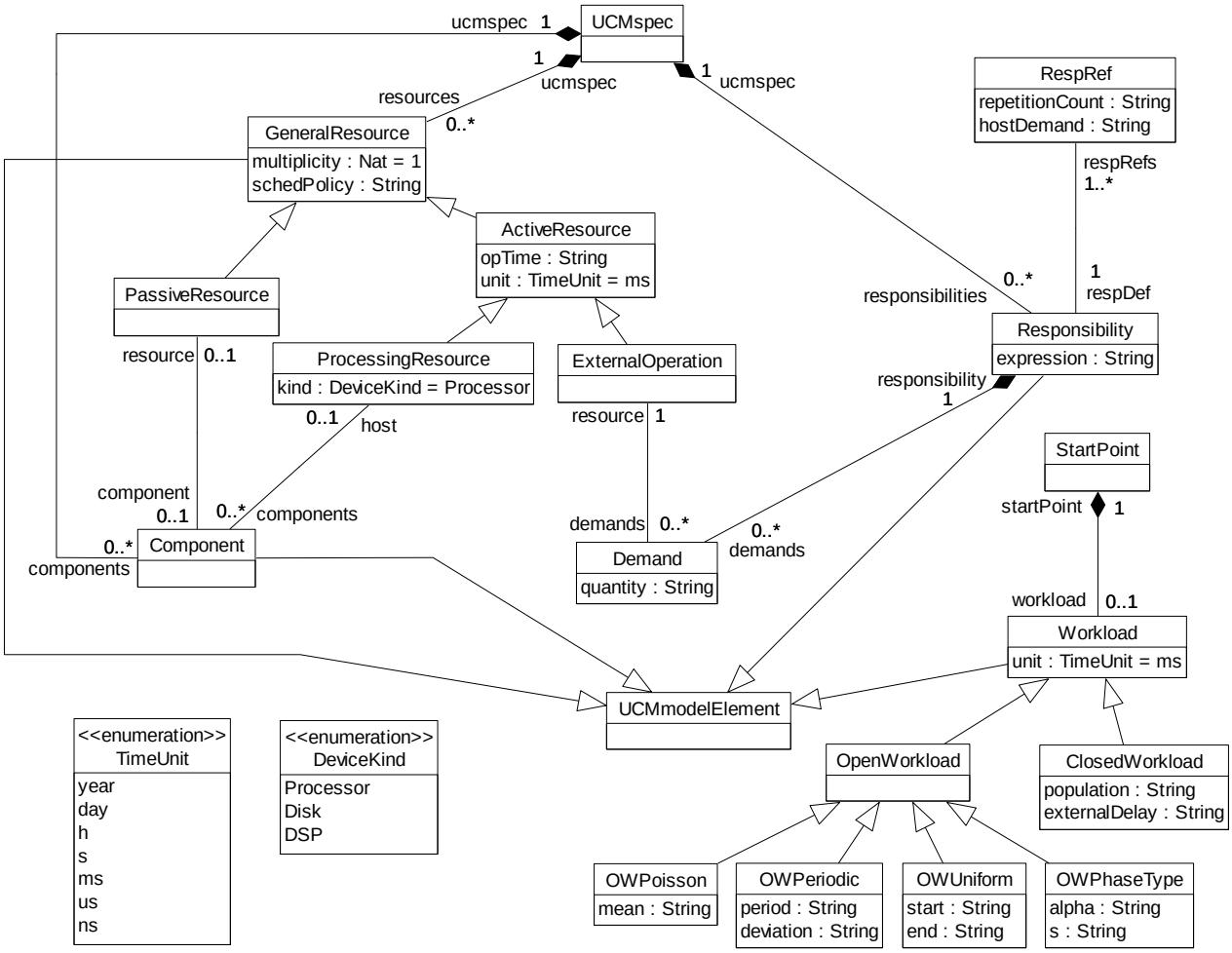


Figure 94 – Abstract grammar: UCM performance overview

I.2 Summary of Concrete Metamodel

Figure 96 shows the top level of the concrete metamodel of URN, which extends the abstract grammar metamodel of Figure 90. 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 90) and all their related concrete metamodel classes in grey color.

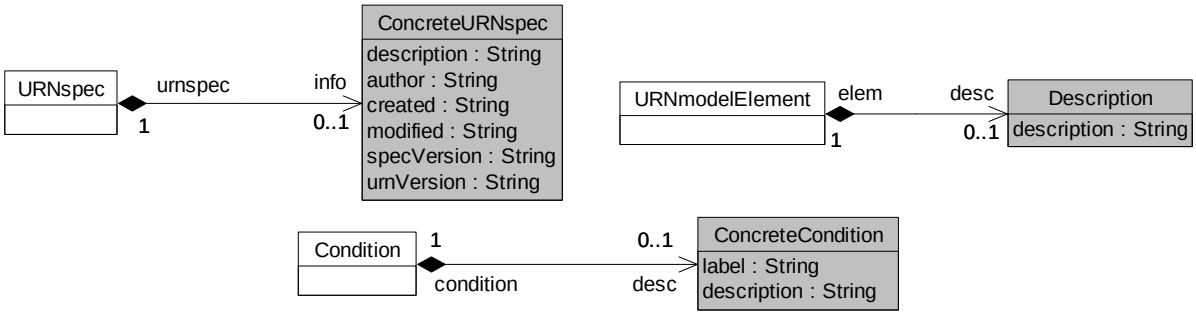


Figure 95 – Concrete grammar: URN top level

Figure 96 presents the complete concrete metamodel of GRL, which extends the abstract grammar metamodel of Figure 90 and Figure 91. The diagram is linked to Figure 90 via *GRLspec*, *URNmodelElement*, and *GRLmodelElement*. It is also linked to Figure 91 via *Actor*, *IntentionalElement*, and *ElementLink*. The diagram shows all concrete metamodel classes in grey color.

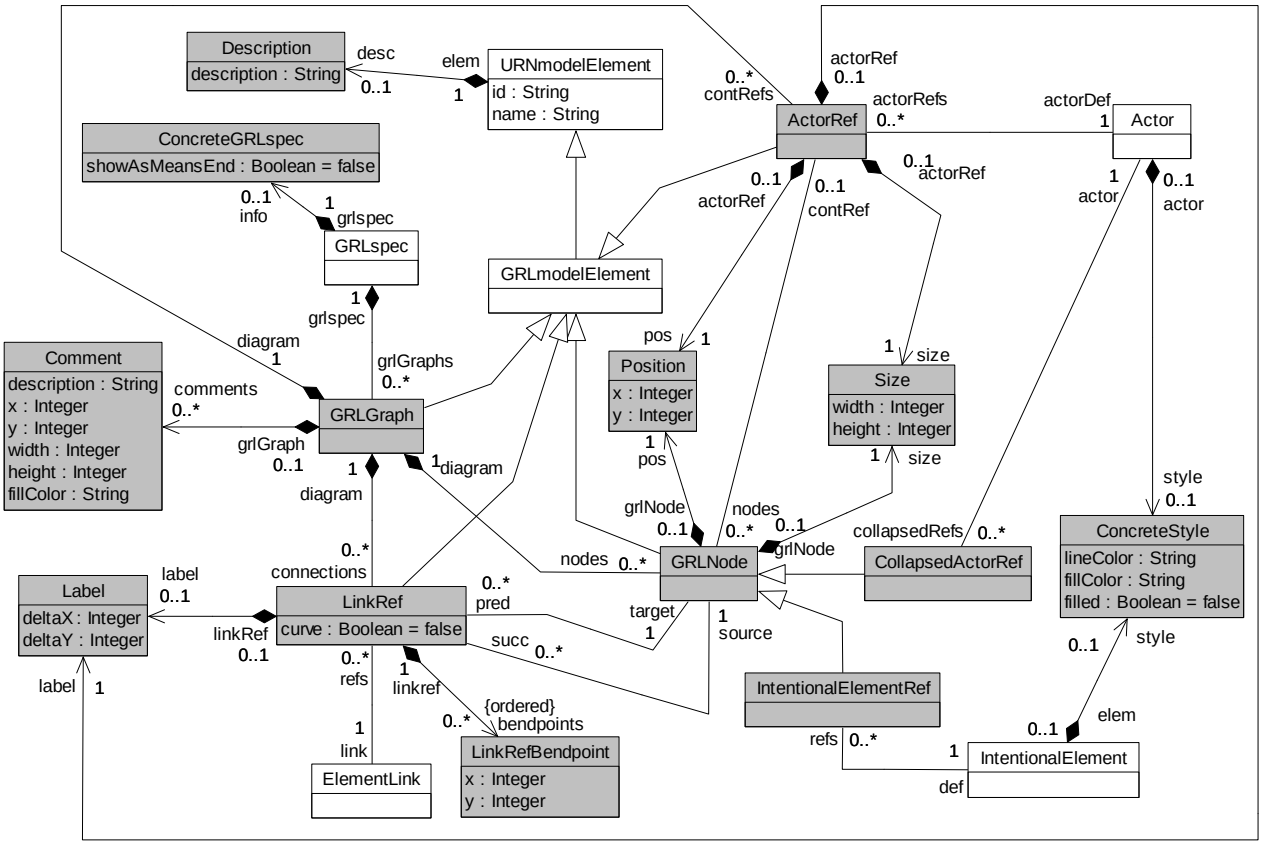


Figure 96 – Concrete grammar: GRL

Figure 97 presents the concrete metamodel of the UCM notation, which extends the abstract grammar metamodel of Figure 90, Figure 92, Figure 93, and Figure 94. The diagram is linked to Figure 90 via *URNmodelElement*, *UCMmodelElement*, and *Condition*. It is linked to Figure 92 via *UCMmap*, *PathNode*, *NodeConnection*, *Component*, and *ComponentRef*. The diagram shows all concrete metamodel classes in grey color.

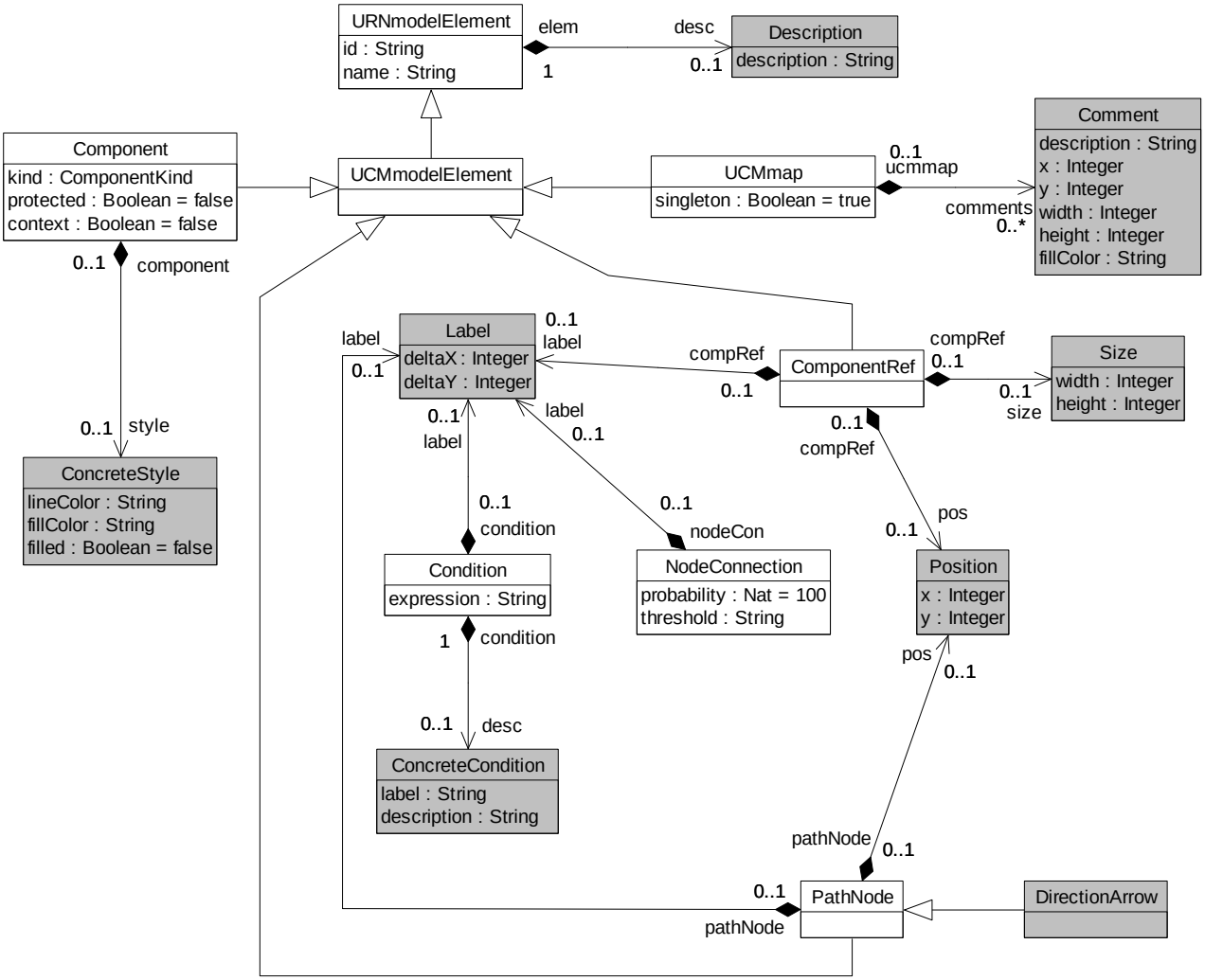


Figure 97 – Concrete grammar: UCM

I.3 Summary of URN Symbols

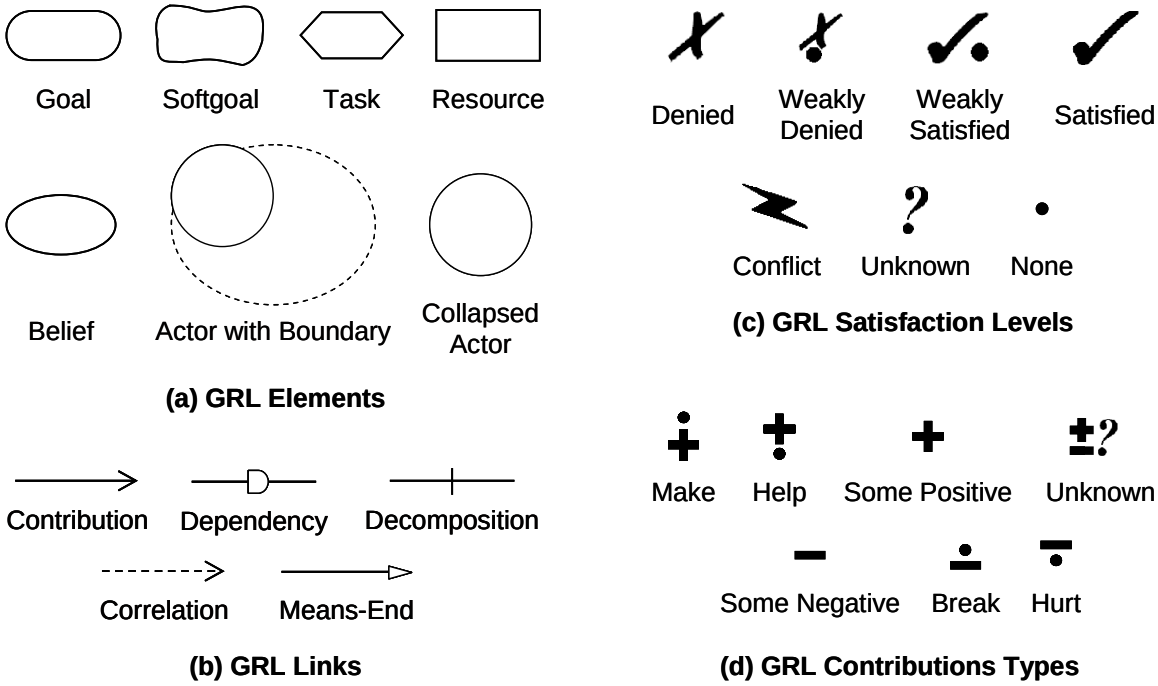


Figure 98 – GRL symbols

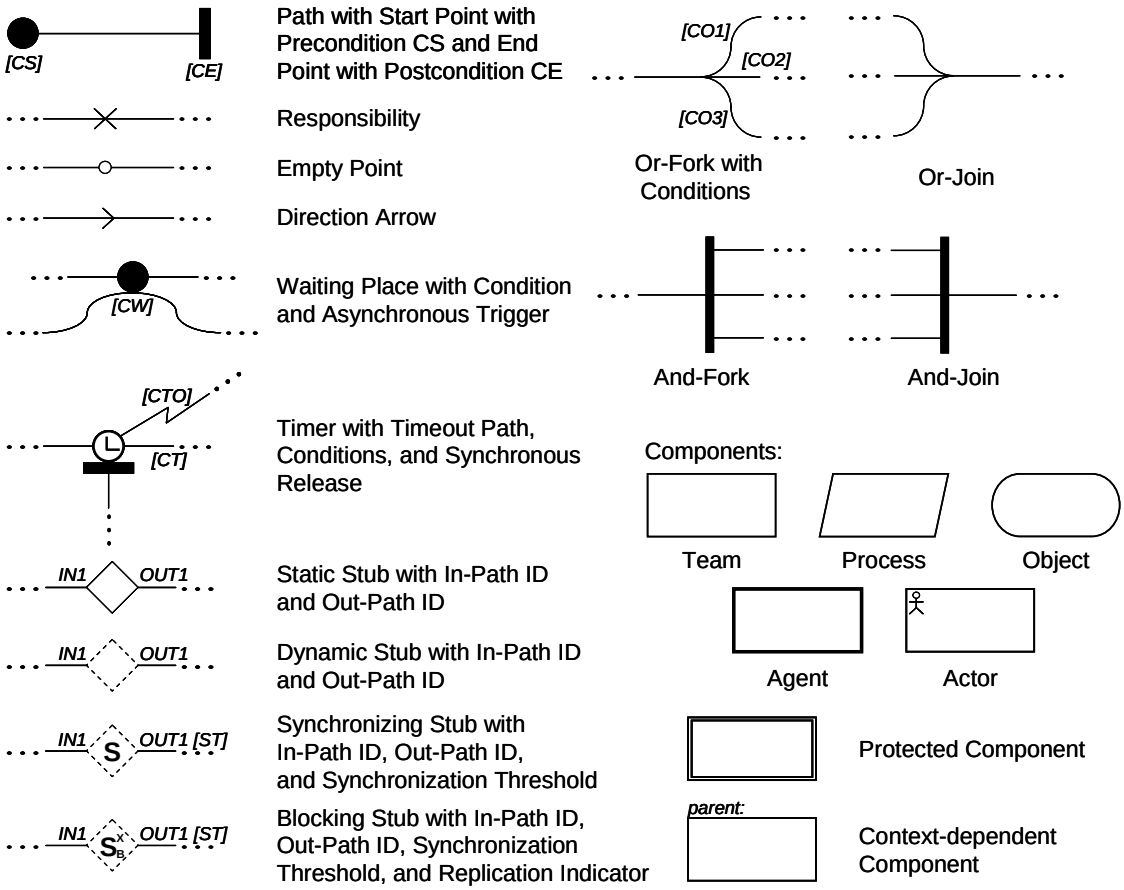


Figure 99 – UCM symbols

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) Actor satisfaction is evaluated.
- (d) Fully automated.
- (e) Cycles in models are handled partially: a cycle will only be evaluated if one of its elements has a value initialized by the strategy.
- (g) Inconsistent evaluation strategies are allowed.
- (h) Evaluations defined as part of a strategy are not overridden.
- (j) 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, (f) no conflict detection, (i) with relation to UCM, and (l) with tolerance.
- Clause II.3: (a) Qualitative evaluation, (f) conflict detection, (i) without relation to UCM, and (l) without tolerance.
- Clause II.4: (a) Hybrid evaluation, (f) no conflict detection, (i) with relation to UCM, and (l) with tolerance.

As for link evaluation functions (k), 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.

II.1.2 Generic algorithm overview

Our example algorithms all follow the same three steps: 1) initialize the evaluation values of the GRL intentional elements 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. 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 100 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. In this algorithm, each intentional element 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 = ∅              // intentional elements that can be evaluated
elementsWaiting:List = ∅            // intentional elements that cannot yet be evaluated

newEvaluations = ∅

for each element:IntentionalElement 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 100 – Example: Forward propagation algorithm

The forward propagation algorithm invokes the calculateEvaluation algorithm (Figure 101), 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:IntentionalElement, 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 101 – Example: Calculate evaluation algorithm

II.2 Example of quantitative evaluation algorithm

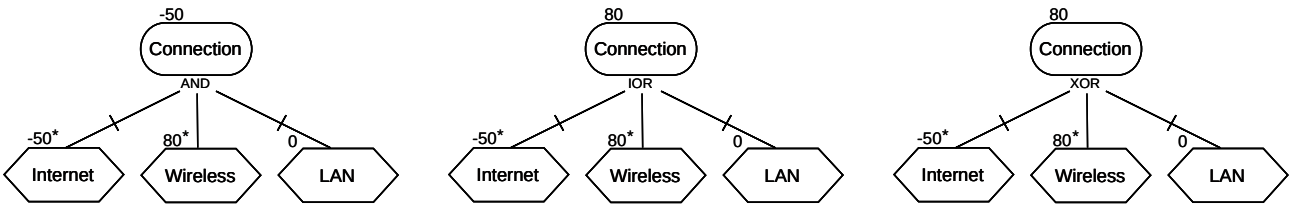
This quantitative GRL algorithm uses *Integer* values for the evaluation, and hence uses the quantitativeContribution attribute of *Contribution*, the importanceQuantitative attribute of *IntentionalElement*s, and the new quantitativeVal attribute of intentional elements initialized from the selected *EvaluationStrategy*.

II.2.1 Calculating quantitative evaluations for decomposition links

This corresponds to the CalculateDecompositions(element) step in Figure 101. The result depends on the type of decomposition (AND, IOR, or XOR).

The satisfaction level of an intentional 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 102 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.



(a) AND decomposition (b) IOR decomposition (c) XOR decomposition

Figure 102 – 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 101. 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:IntentionalElement, 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 if ((contribValue ≤ -100 + tolerance) and not(hasDeny))
        if (totalCont < 0) // negative contribution
                contribValue = min (decompValue, -100 + tolerance)       // case B

return contribValue
```

Figure 103 – Example: Quantitative CalculateContributions algorithm

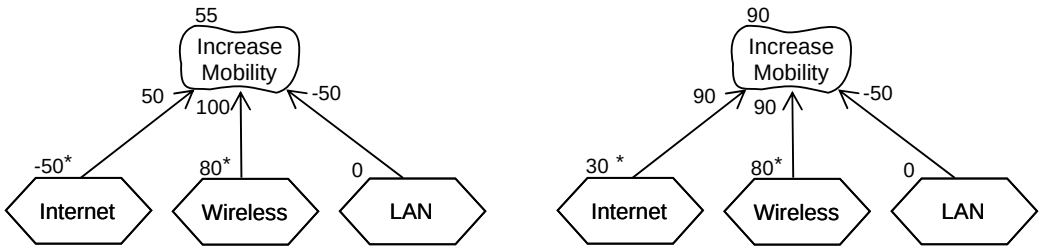
The algorithm in Figure 103 ensures that the satisfaction level of each intentional element will not go above 100 or below –100. In addition, the algorithm takes tolerance into account to ensure that the evalutation value of an intentional element can be 100 (respectively –100) only if a) at least one of the intentional 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 103 and illustrated for positive values in Table 15 (negative values are handled analogously).

NOTE – The quantitative propagation algorithm resolves conflicts.

Table 15 – Example: Calculating contribution values with different tolerance values

Case in Figure 103	*hasSatisfy*	*decompValue*	*totalCont*	*tolerance limit*	*contribValue*
A	false	95	3	100 – 10 = 90	max (decompValue, 90) = 95
hasSatisfy	true				95 + 3 = 98
below tolerance limit	false			100 – 1 = 99	95 + 3 = 98
B	false	95	–3	100 – 0 = 90	95 – 3 = 92
hasSatisfy	true				95 – 3 = 92
below tolerance limit	false			100 – 4 = 96	95– 3 = 92
A	false	85	13	100 – 8 = 92	max (decompValue, 92) = 92
hasSatisfy	true				85 + 13 = 98
below tolerance limit	false			100 – 1 = 99	85 + 13 = 98

Figure 104 provides two examples with three contributions each (the initial decompValue is 0). Strategies initialize two elements. In (a), we have ((–50 × 50) + (80 × 100) + (0 × –50))/100 = 55. In (b), where the tolerance has been set to 10, we have ((30 × 90) + (80 × 90) + (0 × –50))/100 = 99. However, as there is no fully satisfied weighted contribution and decompValue is not 100, then 100-tolerance = 90 is output.



(a) Contributions (b) Contributions with a tolerance of 10

Figure 104 – 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 101. In this algorithm, the source element of the dependency links cannot have an evaluation value higher than those of the intentional 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 105, 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.

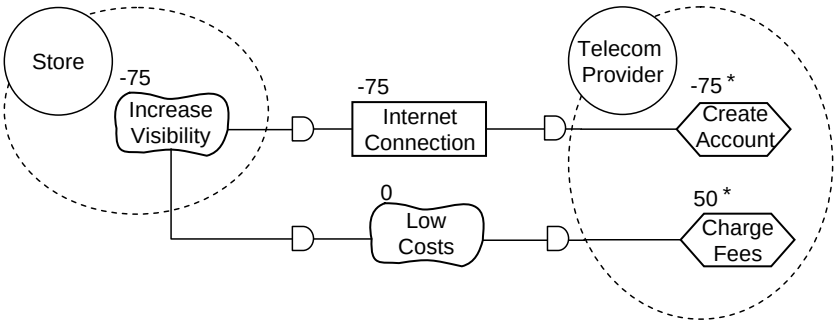


Figure 105 – Example: Quantitative evaluation of dependency links

II.2.4 Calculating quantitative evaluations for actors

This is the third and last step discussed in clause II.1.2. In order to compute the quantitative evaluation value of an actor, it is necessary to identify the quantitative satisfaction value and quantitative importance value of each intentional 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:

$$actor.quantitativeVal = (\sum_{i=1}^{n} elem_i.quantitativeVal \times elem_i.importanceQuantitative) / 100 / n$$

For example, Figure 106 shows an actor with four softgoals, three of which with non-zero importance. The quantitative value of the actor's satisfaction becomes:

$$((100 \times 100) + (100 \times 29) + (-75 \times 60)) / 100 / 3 = 28$$

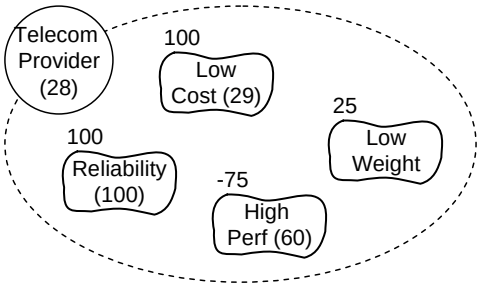


Figure 106 – Example: Quantitative evaluation of actors

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 *IntentionalElement*s, and the new attribute qualitativeVal of intentional 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 7.3.3) are *High*, *Medium*, *Low* and *None*. Since these values are discrete, the propagation algorithm must consider them individually. To this end, lookup tables and partial orderings are often used to define necessary functions explicitly.

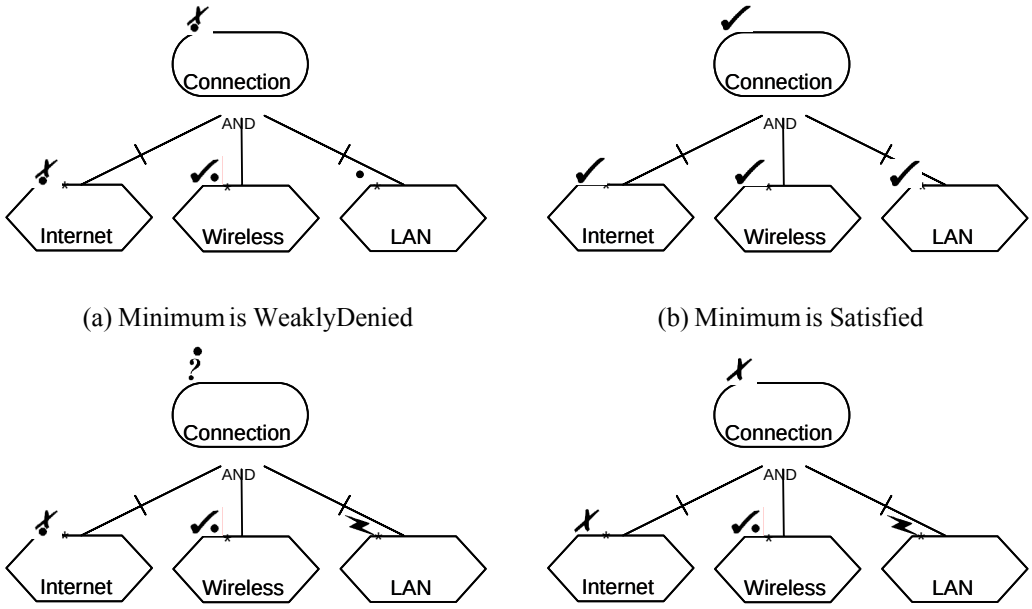
II.3.1 Calculating qualitative evaluations for decomposition links

This corresponds to the CalculateDecompositions(element) step in Figure 101. The result depends on the type of decomposition (AND, IOR, or XOR).

The satisfaction level of an intentional 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 107 provides four examples of qualitative AND-type decomposition that illustrate this propagation.



(a) Minimum is WeaklyDenied

(b) Minimum is Satisfied

(c) Minimum is Conflict: Undecided is propagated

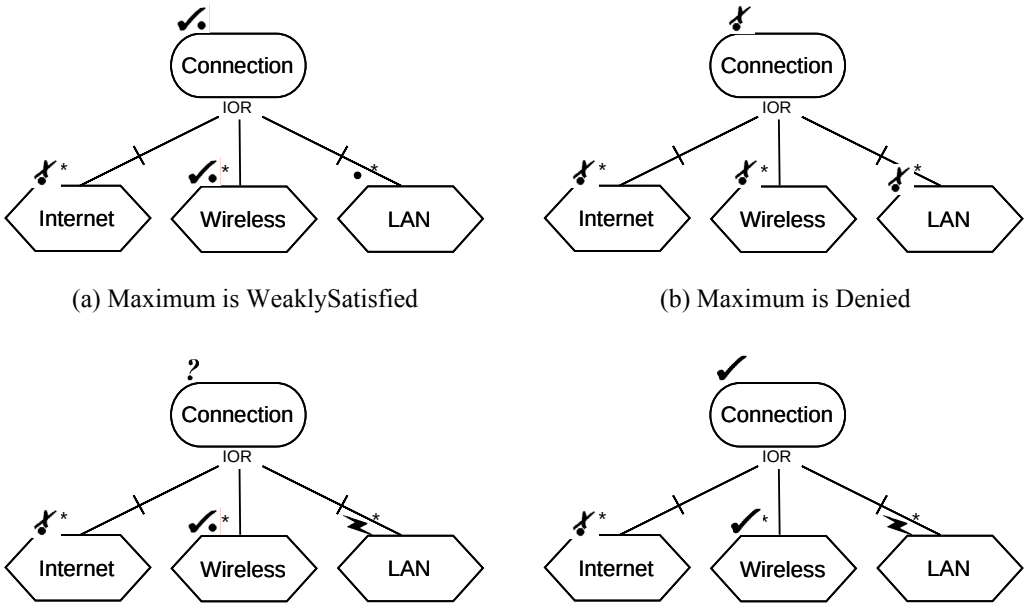
(d) Minimum is Denied, even if Conflict is present

Figure 107 – 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 108 provides four examples of qualitative IOR-type decomposition that illustrate this propagation.



(a) Maximum is WeaklySatisfied (b) Maximum is Denied

(c) Maximum is Conflict: Undecided is propagated (d) Maximum is Satisfied, even if Conflict is present

Figure 108 – 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 101. 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 109.

```
Algorithm CalculateContributions
Inputs element:IntentionalElement, 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(oneCount, 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 109 – Example: Qualitative CalculateContributions algorithm

The AdjustContributionCounters algorithm (Figure 110) 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 110 – Example: AdjustContributionCounters algorithm

The CalculateContributions algorithm also uses a WeightedContribution function that computes one qualitative weighted contribution according to the lookup table in Table 16, 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 16 – WeightedContribution function for the computation of one weighted contribution

	Make	***Help***	***SomePositive***	***Unknown***	***SomeNegative***	***Hurt***	***Break***
Denied	Denied	WeaklyDenied	WeaklyDenied	None	WeaklySatisfied	WeaklySatisfied	Satisfied
WeaklyDenied	WeaklyDenied	WeaklyDenied	WeaklyDenied	None	WeaklySatisfied	WeaklySatisfied	WeaklySatisfied
WeaklySatisfied	WeaklySatisfied	WeaklySatisfied	WeaklySatisfied	None	WeaklyDenied	WeaklyDenied	WeaklyDenied
Satisfied	Satisfied	WeaklySatisfied	WeaklySatisfied	None	WeaklyDenied	WeaklyDenied	Denied
Conflict	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown
None	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* returned. Formally:

CompareSatisfiedAndDenied (ns, nd) = *Conflict*, **if** (ns > 0 **and** nd > 0)
= *Satisfied*, **if** (ns > 0 **and** nd = 0)
= *Denied*, **if** (nd > 0 **and** ns = 0)
= *None*, **if** (ns = 0 **and** nd = 0)

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:

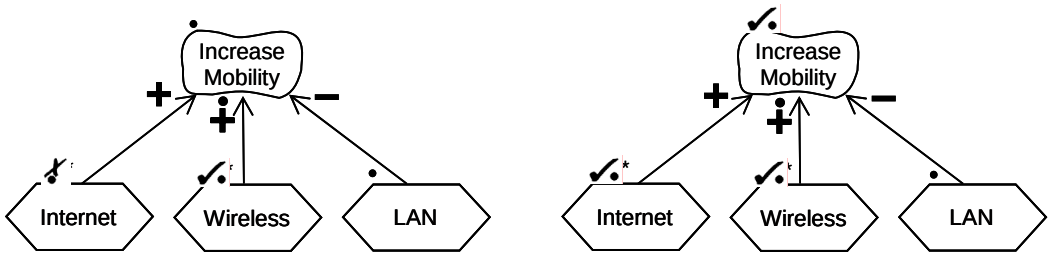
CompareWSandWD (ws, wd) = *WeaklySatisfied*, **if** (nws > nwd)
= *WeaklyDenied*, **if** (nwd > nws)
= *None*, **if** (nwd = nws)

The final result is computed with the CombineContributions function, which combines the previously computed values according to Table 17. 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 17 – CombineContributions function for the computation of the final contribution

	Denied	***Satisfied***	***Conflict***	***None***
Weakly Denied	Denied	Weakly Satisfied	Conflict	Weakly Denied
Weakly Satisfied	Weakly Denied	Satisfied	Conflict	Weakly Satisfied
None	Denied	Satisfied	Conflict	None

Figure 111 provides two examples with three contributions each (the initial decompValue is *None*). Strategies initialize two elements in each example. In (a), we have (*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), we have (*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*.



(a) Contributions: None is propagated (b) Contributions: WeaklySatisfied is propagated

Figure 111 – Example: Qualitative evaluation of contribution links

II.3.3 Calculating qualitative evaluations for dependency links

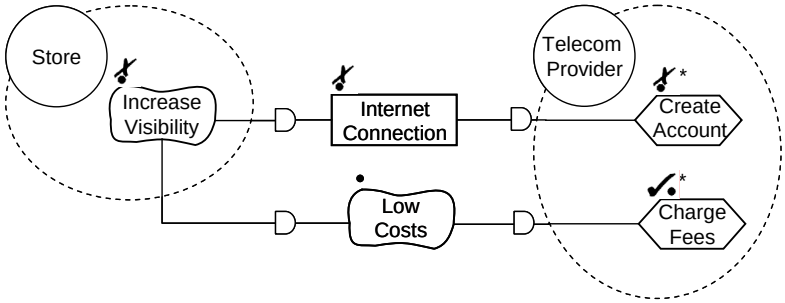
This corresponds to the CalculateDependencies(element, contribValue) step in Figure 101. In this algorithm, the source element of the dependency links cannot have an evaluation value higher than those of the intentional elements it depends on (i.e., the target elements of the dependency links). This 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*

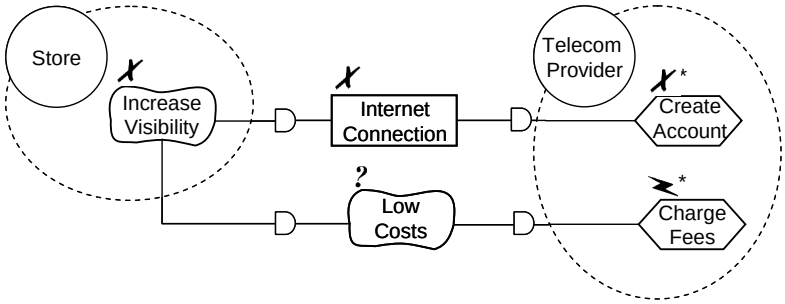
Again, *Conflict* results are substituted with *Unknown* as conflicts are not propagated.

Two examples are shown in Figure 112, 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 105. 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



(b) Minimum is Denied, even if Conflict is present

Figure 112 – Example: Qualitative evaluation of dependency links

II.3.4 Calculating qualitative evaluations for actors

This is the third and last step discussed in clause II.1.2. In order to compute the qualitative evaluation value of an actor, the qualitative satisfaction value and qualitative importance value of each intentional element bound to the actor are used.

The CalculateActorEvaluation algorithm is similar to the qualitative CalculateContributions algorithm (Figure 109) and reuses some of its sub-algorithms. First, the qualitative evaluation value of each intentional element bound to the actor is weighted according to the importance of that element to the actor. This WeightedImportance function is defined in Table 18, 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 110) 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:IntentionalElement 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 113 – Example: Qualitative CalculateActorEvaluation algorithm

Table 18 – WeightedImportance function for the computation of one element value

	Denied	***WeaklyDenied***	***WeaklySatisfied***	***Satisfied***	***Conflict***	***Unknown***	***None***
High	Denied	WeaklyDenied	WeaklySatisfied	Satisfied	Conflict	Unknown	None
Medium	WeaklyDenied	WeaklyDenied	WeaklySatisfied	WeaklySatisfied	Conflict	Unknown	None
Low	WeaklyDenied	None	None	WeaklySatisfied	Conflict	Unknown	None
None	None	None	None	None	None	None	None

For example, Figure 114 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*.

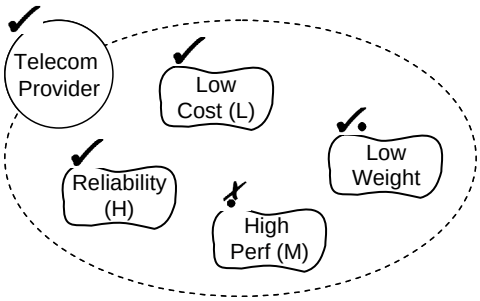


Figure 114 – Example: Qualitative evaluation of actors

II.4 Example of hybrid evaluation algorithm

This hybrid GRL algorithm uses *Integer* values for the evaluation, and hence uses the importanceQuantitative attribute of *IntentionalElement*s and the new quantitativeVal attribute of intentional 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 quantitative 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 101. The algorithm is the same as the quantitative algorithm in clause II.2.1.

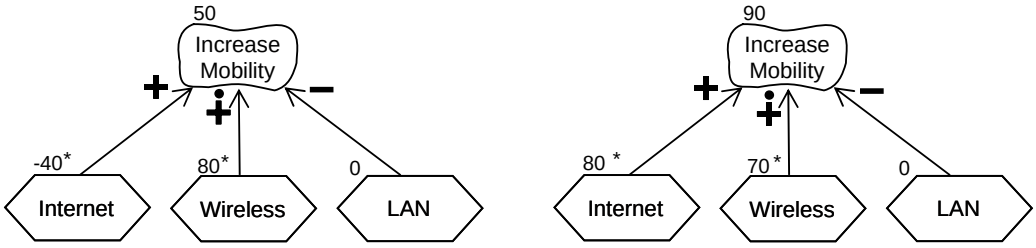
II.4.2 Calculating hybrid evaluations for contribution links

This corresponds to the CalculateContributions(element, decompValue) step in Figure 101. The algorithm first maps all qualitative contributions to quantitative contributions using Table 19. 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 19 – Quantitative contribution values for qualitative contributions

Qualitative Contribution	*Quantitative Contribution*
Make	100
SomePositive	75
Help	25
Unknown	0
Hurt	–25
SomeNegative	–75
Break	–100

Figure 115 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 19. In (a), we have ((–40 × 75) + (80 × 100) + (0 × –75))/100 = 50. In (b), where the tolerance has been set to 10, we have ((80 × 75) + (70 × 100) + (0 × –75))/100 = 130. However, as there is no fully satisfied weighted contribution, then 100-tolerance = 90 is output.



(a) Contributions (b) Contributions with a tolerance of 10

Figure 115 – 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 101. The algorithm is the same as the quantitative algorithm in clause II.2.3.

II.4.4 Calculating hybrid evaluations for actors

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.

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 55. 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, sequenceNumber 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 116. 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 > 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
                        current.sequenceNumbers.add(sequenceNumber);
                        sequenceNumber++;
                }

                // if possible, move the first enabled path node to the current path nodes
                if (enabledPathNodes.size() > 0)
                {
                        // remove the first element from the list of enabled path nodes
                        nextCurrent = enabledPathNodes.removeFirstElement()

                        // not shown here but this is the place where any additional behaviour
                        // for the nextCurrent path node is executed depending on the type
                        // of path node; this may but is not limited to raising further traversal
                        // exceptions, removing the same path node as nextCurrent from
                        // currentPathNodes or blockedPathNodes in case of synchronization
                        // (also adjusts the visitNumber accordingly), and resolving
                        // component plug-in bindings

                        // add the nextCurrent path node at the beginning of the list
                        // of currently visited path nodes, so that the traversal continues
                        // in a depth-first way
                        currentPathNodes.addAtBeginning(nextCurrent)
                        nextCurrent.visitNumber++
                        if (nextCurrent.visitNumber > maxVisit)
                                throw new TraversalException("warning: infinite loop")

                        // since moving to a new path node may impact already blocked
                        // path nodes, the blocked path nodes are added back to the currently
                        // visited path nodes; however, they are added at the end of the list
                        // to ensure a depth-first approach
                        currentPathNodes.addAllAtEnd(blockedPathNodes)
                        blockedPathNodes.clear()
                }
        }
}
catch (TraversalException te)
{
        warningsAndErrors.add(te.getMessage())
}
// verify successful completion of traversal
for each pn:PathNode in (currentPathNodes ∪ 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 116 – Example: Depth-first UCM path traversal mechanism

Given the depth-first algorithm from Figure 116, 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 = ∅, blockedPathNodes = ∅, 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 = ∅ (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 which 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: currentPathNodes = {ring}, blockedPathNodes = {ringing}.
- try next path node in currentPathNodes; ring end point 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 traversal of the "TL pin TCS success" scenario is shown in Figure 117. 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.

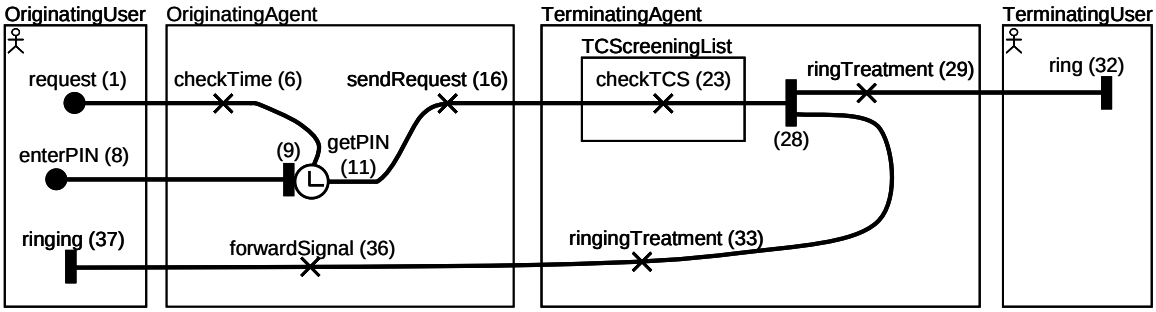


Figure 117 – Example: Flattened UCM model of depth-first traversed scenario

III.3 Example of breadth-first UCM path traversal mechanism

The difference between the depth-first approach shown in Figure 116 and the breadth-first approach shown in Figure 118 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 must be 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 ∪ 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 118 – Example: Breadth-first UCM path traversal mechanism

Given the breadth-first algorithm from Figure 118, 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 the 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 117, except for the sequence numbers. The difference can be seen in Figure 119 for the path nodes that follow the AND-fork.

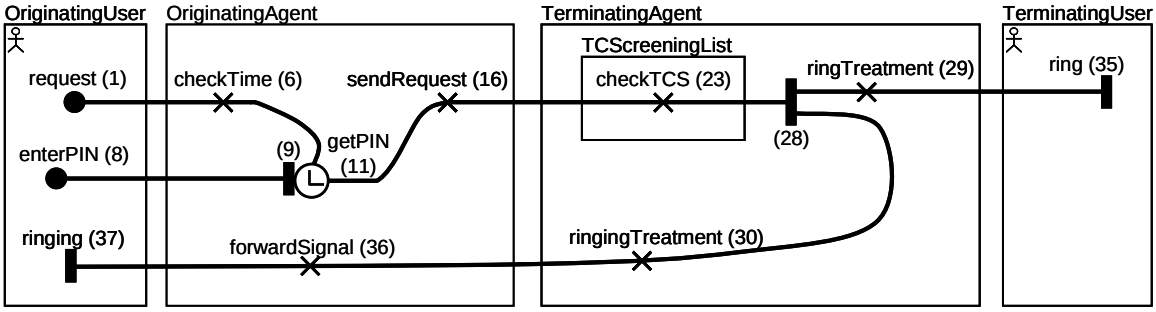


Figure 119 – Example: Flattened UCM model of breadth-first traversed scenario

URN Change Request Form

Please fill in the following details		
Character of change:	❑ error correction	❑ clarification
	❑ simplification	❑ extension
	❑ modification	❑ decommission
Short summary of the change request		
Short justification of the change request		
Have you consulted other users	❑ yes	❑ no
Is this view shared in your organization	❑ yes	❑ no
How many users do you represent?	❑ 1-5	❑ 6-10
	❑ 11-100	❑ over 100
Your name and address		

Please attach further sheets with details if necessary

URN (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.

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/, last accessed July 2009.

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	General tariff principles
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	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	Terminals and subjective and objective assessment methods
Series Q	Switching and signalling
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 and next-generation networks
Series Z	**Languages and general software aspects for telecommunication systems**

Printed in Switzerland
Geneva, 2010