CAN COM 10-12

November 2000: English

**UIT - Secteur de la normalisation des télécommunications**

**ITU - Telecommunication Standardization Sector**

**UIT - Sector de Normalización de las Telecomunicaciones**

| | | | |
|---|---|---|---|
| Commission d'études | ) | Contribution tardive | ) |
| Study Group | ) 10 | Delayed Contribution | ) **D.____** |
| Comisión de Estudio | ) | Contribución tardia | ) |

Geneva, 21 November - 24 November 2000

| | | |
|---|---|---|
| Texte disponible seulement en | ) | |
| Text available only in | ) | **E** |
| Texto disponible solamente en | ) | |

---

Question: 12/10

Source*: CANADA

Title**:** **Draft Specification of the User Requirements Notation**

---


**ABSTRACT**

At its meeting in November 1999, Study Group 10 approved Question 12: URN: User Requirements Notation to study what new Recommendations and other documents are required in order to define a notation, methods or revised notations for capturing and analyzing user requirements. Subsequently, Question 12 was approved by TSAG at its meeting in Montreal, in September 2000.

The attached document entitled, *Draft Recommendation Z.URN: Languages for Telecommunication Applications - User Requirements Notation*, to this contribution is the first draft specification of the User Requirements Notation. This specification meets a number of criteria for the required notation specified in the definition of Question 12. It is based on the experience with the notation called Use Case Maps (UCMs) presented at the November 1999 meeting of Study Group 10. This will contribute to one of the task objectives of Question 12: to study or possibly revise existing notations for the purpose of defining the required notation.


**PROPOSAL**

It is proposed that the attached document be used as the starting base document for Question 12.

---


* Contact: - Don Cameron,
Nortel Networks Corporation

Tel: +1 613 763 4486
Fax: +1 613 763 6681
e-mail: dcameron@nortelnetworks.com

TELECOMMUNICATION UNION

**TELECOMMUNICATION STANDARDIZATION SECTOR**

**COM 10-XX-E**

**November 2000**

**Original: English**

STUDY PERIOD 1997-2000

**STUDY GROUP 10 - CONTRIBUTION XX**

SOURCE *: RAPPORTEUR, QUESTION 12/10

TITLE: DRAFT RECOMMENDATION Z.URN: LANGUAGES FOR TELECOMMUNICATION APPLICATIONS - USER REQUIREMENTS NOTATION

---

At its meeting in November 1999, Study Group 10 approved Question 12: URN: User Requirements Notation to study what new Recommendations and other documents are required in order to define a notation, methods or revised notations for capturing and analyzing user requirements. Subsequently, Question 12 was approved by TSAG at its meeting in Montreal, in September 2000.

This contribution is the first draft specification of the User Requirements Notation. This specification meets a number of criteria for the required notation specified in the definition of Question 12. It is based on the experience with the notation called Use Case Maps (UCMs) presented at the November 1999 meeting of Study Group 10. It is intended to contribute to one of the task objectives of Question 12: to study or possibly revise existing notations for the purpose of defining the required notation.

---

*** Contact:** - Don Cameron,
Nortel Networks Corporation

Tel: +1 613 763 4486
Fax: +1 613 763 6681
e-mail: dcameron@nortelnetworks.com

TABLE OF CONTENTS

## FOREWORD

ITU (International Telecommunication Union) is the United Nations Specialized Agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of the ITU. The 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) meets every four years to establish the topics for study by the ITU-T Study Groups, which produce Recommendations on these topics.

The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSA Resolution No. 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 term *recognized operating agency (ROA)* includes any individual, company, corporation or governmental organization that operates a public correspondence service. The terms *Administration, ROA* and *public correspondence* are defined in the *Constitution of the ITU (Geneva, 1992).*

## INTELLECTUAL PROPERTY RIGHTS

The ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. The 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, the ITU had/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.

NEW RECOMMENDATION Z.URN: LANGUAGES FOR TELECOMMUNICATIONS APPLICATIONS - USER REQUIREMENTS NOTATION

# SUMMARY


*Scope-objective*

This Recommendation defines URN (*User Requirements Notation*) intended for describing user requirements scenarios in a formal way without any reference to implementation mechanisms and with optional dependency on component specification. Such a notation is needed to capture user requirements prior to any design.

*Coverage*

URN has concepts for the specification of behaviour, structuring, and non-functional requirements. This document presents requirements for URN, textual and graphical representations of URN constructs, and an assessment of conformity of the current URN representations to the requirements for URN.

*Applications*

URN is applicable within standard bodies and industry. The main applications areas, which URN has been designed for are stated in Section 2.8, but URN is generally suitable for describing reactive systems. The range of application is from requirement description to high-level design.

*Status/Stability*

This Recommendation is a draft reference manual.

The main text is accompanied by annexes and appendices:

- Annex A Document type definition
- Annex B Tutorial
- Annex C Relationships between URN and other notations
- Appendix I Requirements engineering activities
- Appendix II Tool issues
- Appendix III External references
- Appendix IV Guidelines for the maintenance of URN

*Associated work*

None are currently defined.

*Keywords*

Data types, formal description technique, functional requirements specification, non-functional requirements specification, graphical notation, hierarchical decomposition, goals, scenarios, specification technique.

# USER REQUIREMENTS NOTATION (URN)

## 1 Introduction

The text of this clause is not normative.

### 1.1 Motivation

A notation is needed that can describe user requirements scenarios without any reference to messages or system components and their states but at the same time can capture the user requirements prior to design. The focus during the requirements specification stage is on behaviour and on quality attributes. The notation can also be used during the high-level design phase when responsibilities specified in the scenarios are allocated to components. Scenario specification without sub-system component reference would facilitate reusability of scenarios across a wide range of architectures. The ability of the notation to straddle requirements specification and high level design will facilitate negotiations between stakeholders and implementers.

There is an increasing demand for non-static protocols with policy-driven negotiation using dynamic entities. Agent-based systems are examples of systems that require policy-driven mechanisms. When specifying such protocols, it is not possible to make an early commitment to messages and components at the requirements capture phase.

There is also the need for detection and avoidance of undesirable interactions between features. Older techniques require large investment in terms of messages and components that need to be checked for interactions. Using the notation specified in this Recommendation can provide insights at the requirements level and enable designers to reason about feature interactions early in the design process.

It is also important to deal with non-functional requirements (NFRs) in a more systematically manner during requirements analysis and during design. NFRs are requirements such as stringent performance constraints, systems operational costs, reliability, maintainability, portability, interoperability, robustness, and the like. In today's software development practice, many NFRs are stated only informally, making them difficult to analyse, specify and enforce during software development, and to be validated by the user once the final system has been built. NFRs, however, do play a crucial role during system development, serving as selection criteria for choosing among myriad of decision during requirements analysis, when, for example, aiding in determining what the system boundaries should be and what functional requirements to include, and during design and implementation when generating and evaluating alternative solutions.

Many of today's approaches to deal with NFRs originate from the technical work related to quality metrics. Such approaches attempt to quantify NFRs, and then measure to what extent an existing software system or parts of it meet the desired non-functional requirements. Particular NFRs such as performance, reliability, software complexity, development process maturity have proposed metrics that are used to measure aspects of an existing software system and/or its development process. These approaches, that use quantitative methods for testing and evaluating NFRs, are particularly useful when dealing with quality aspects of very large and complex software systems, and for quality related process improvement programs, but also to reason about hard (performance) constraints the system needs to met. Other approaches which recognise that many NFRs are often difficult, if not impossible, to quantify, use qualitative oriented methods (such as architectural change scenarios) to evaluate software systems, or combinations of both (qualitative and quantitative methods). These approaches, however, assume an already existing software system (or parts thereof) that is evaluated for its NFR properties. They do not assist in the specification of NFRs prior to building the system, nor do they provide support during the analysis and design of software systems. This notation deals with NFRs during the process of analysis and design, when most NFRs are addressed and satisfied, trade-offs between conflicting NFRs are made and their rationale captured.

The User Requirements Notation (URN) and methodology defined in this Recommendation are related to SDL, MSC, UML and TTCN and their methodologies. Additionally, URN is defined to have the following capabilities:

a) describe scenarios without reference to messages, system components, or component states;

b) capture user requirements when very little design detail is available;

c) facilitate reusability of scenarios across a wide range of architectures with allocation of scenario responsibilities to architectural components;

d) have dynamic refinement capability with the ability to allocate scenario responsibilities to architectural components;

e) be applicable to the design of policy-driven negotiation protocols involving dynamic entities;

f) facilitate detection and avoidance of undesirable interactions between features;

g) provide insights at the requirements level to enable designers to reason about feature interactions and performance trade-offs early in the design process.

h) provide facilities to express, analyse and deal with non-functional requirements;

i) provide facilities to express the relationship between business objectives and goals to system requirements expressed as scenarios and global constraints over the system, its development, deployment, maintenance and evolution and operational processes;

j) provide facilities to capture reusable analysis and design knowledge related to know-how for addressing non-functional requirements.

The current methods that used informal natural language for capturing requirements can leave too much open for interpretation and can contain invalid logic. Manual methods are used to validate these specifications with the result that defects are sometimes not caught until the implementation phase. Studies of software development have clearly shown that the earlier defects are detected, the less costly they are to repair. Standardization of requirements engineering activities aims to make it easier to detect more defects at the requirements definition stage.

These same informal methods have also proven less than satisfactory for negotiating priorities among competing business objectives and in general for managing trade-offs in the domain of non-functional requirements. The end result can be the delivery of a product to market that does not satisfy customers and does not meet business objectives. Standardization of requirements engineering activities aims to make it easier to define a product that balances stakeholder objectives and satisfies customer expectations.

Standardization of a formally defined notation used for capturing user requirements is a move to make the practice of this activity more rigorous and predictable and the results yielded by this activity clearer, more consistent, correct, and complete. These results should lead to reduced development costs, earlier delivery of product to market, and increased customer satisfaction.

## 1.2 Document organization

The document first defines the scope of this standard. The scope of the URN is presented in Section 2. This section:

- gives an overview of the requirements engineering activity that uses the URN
- gives an overview of the activities that are prerequisite and subsequent to the URN activity
- discusses what artifacts are input to the URN activity and what attributes these artifacts must possess
- discusses what artifacts are output from the URN activity and what attributes these artifacts must possess
- discusses why traceability among all of the artifacts developed by these activities is desirable, and how traceability can be achieved
- describes the flow of the URN activity, how constraints affect that flow, and what URN capabilities are used for what purpose
- discusses the relationships between functional and non-functional requirements and how these relationships are reflected in the URN; this includes discussion of traceability between non-functional and functional requirements definition within the URN
- defines a set of capabilities the URN should possess; this set of capabilities includes ones that have not yet been realized and so are goals; rationales are provided for requiring these capabilities

Section 3 contains formal definitions of the top-level elements of URN as well as several conventions.

Section 4 contains formal definitions of the graphical and textual elements of the Non-Functional Requirements URN (URN-NFR).

Section 5 contains formal definitions of the graphical and textual elements of the Functional Requirements URN (URN-FR). It is also intended to include the definition of the execution semantics of the URN-FR.

Section 6 contains a list of the requirements that the URN must fulfill together with statements concerning the current status of the notation with respect to the requirements.

Section 7 is a specification of the tests that an implementation of the URN must satisfy in order to be deemed to be in conformance with this standard.

Section 8 lists issues for further study. For example, one issue is defining formal relationships between the URN-FR and the URN-NFR. Some indication of priority is given.

Three annexes are defined:

- Annex A contains a visual representation of the URN Document Type Definition (DTD) together with tables enumerating its elements and attributes. Basic information on how to read XML DTDs is also provided.
- Annex B contains tutorials for the URN-FR and URN-NFR.
- Annex C describes relationships that have been defined between the URN and other notations, both upstream (more abstract) and downstream (more concrete) notations.

Four appendices are defined:

- Appendix I presents general activities in requirements engineering.
- Appendix II deals with tool issues. Prototype tools developed at universities exist for both the URN-FR and the URN-NFR. This appendix details the experience gained from implementing the tools and getting feedback from users.
- Appendix III lists external references to books, journal papers and conference papers.
- Appendix IV describes proposed guidelines for the maintenance of URN.

## 1.3 Related documents and Recommendations

The URN recommendation is related to the following standards and recommendations:

- ISO, IS 8807: Information Processing Systems, Open Systems Interconnection, *LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*.
- ITU-T, Q.1200 General Series, *Intelligent Networks Recommendation Structure*.
- ITU-T, Recommendation Z.100, *Specification and Description Language*.
- ITU-T, Recommendation Z.105, *SDL Combined with ASN.1*.
- ITU-T, Recommendation Z.109, *SDL combined with UML*.
- ITU-T, Recommendation Z. 120: *Message Sequence Chart*.
- ITU-T, Draft revised Recommendation X.292/Z.140: *Methods for testing and specification (MTS) and The tree and tabular combined notation version 3 - TTCN-3: Core language*.
- OMG, *Meta Object Facility Specification (MOF)*, version 1.3.
- OMG, *XML Metadata Interchange Specification (XMI)*, version 1.1
- OMG - UML RTF, *Unified Modeling Language Specification (UML)*, version 1.3.
- W3C Recommendation, *Extensible Markup Language (XML) 1.0* (Second Edition).

## 1.4 Abbreviations

| | |
|---|---|
| *ADT* | Abstract Data Type |
| *ANSI* | American National Standard Institute |
| *ASN.1* | Abstract Syntax Notation One |
| *DTD* | Document Type Definition |
| *ETSI* | European Telecommunication Standards Institute |
| *FDT* | Formal Description Technique |
| *FR* | Functional Requirements |
| *GRL* | Goal-oriented Requirement Language |
| *IN* | Intelligent Network |
| *IP* | Internet Protocol |
| *ISO* | International Organisation for Standardization |
| *ITU* | International Telecommunications Union |
| *LOTOS* | Language Of Temporal Ordering Specification |
| *MOF* | Meta Object Facility |
| *MSC* | Message Sequence Chart |

| | |
|---|---|
| *NFR* | Non-Functional Requirements |
| *OMG* | Object Management Group |
| *SDL* | Specification and Description Language |
| *TTCN* | Tree and Tabular Combined Notation |
| *UCM* | Use Case Map |
| *UML* | Unified Modelling Language |
| *URN* | User Requirements Notation |
| *URN-FR* | User Requirements Notation — Functional Requirements |
| *URN-NFR* | User Requirements Notation — Non-Functional Requirements |
| *W3C* | World Wide Web Consortium |
| *XMI* | XML Metadata Interchange |
| *XML* | eXtensible Markup Language |

# 2 Scope

This section contains an overview of the URN. More detail is provided in the sections where the notations are formally defined and in the section containing the compliance statement (Sections 3 to 6).

## 2.1 What is a URN?

The User Requirements Notation (URN) allows stakeholders to specify, review for correctness, and even discover requirements for a proposed new system or for extensions to an existing system. The URN is intended for use in requirements descriptions in specifications developed by national and international standards organizations. In ITU, requirements descriptions are often called *Stage 1* descriptions. The URN is also intended for use by commercial organizations developing requirements specifications for new products and product extensions; these specifications are not necessarily governed by standards.

The URN is used to construct functional and non-functional requirements models. This standard shall specify a functional requirements URN (URN-FR) and a non-functional requirements URN (URN-NFR) as well as a set of relationships between the URN-FR and the URN-NFR.

The URN is viewed as complementary to notations such as Message Sequence Charts (MSC), the Specification and Description Language (SDL), TTCN-3, and the Unified Modelling Language (UML). Information contained in URN models could possibly be exported to these other notations.

### 2.1.1 What is an URN-NFR?

Business objectives and product quality attributes are modelled using the URN-NFR. The stakeholders use the URN-NFR model to identify and negotiate trade-offs among competing objectives and quality attributes. An outcome of this exercise is a set of technology and implementation choices that reflect these trade-offs. The outcome of the URN-NFR modelling exercise sets the context for the URN-FR modelling exercise.

Non-functional requirements (NFRs, also called quality requirements) are global requirements on the software system, its development, deployment, maintenance and evolution and operational processes, such as the systems operational costs, performance, reliability, maintainability, portability, robustness, and the like. NFRs may originate from objectives related to the business organisation (sometimes called consumer-oriented quality requirements) but also from requirements on the software system, its development environment and process (sometimes called technically-oriented quality requirements). Errors of omission or commission in laying down and taking properly into account such requirements are generally acknowledged to be among the most expensive and difficult to correct once a software system has been implemented, and have direct impact on the success of the software system. NFRs are difficult to specify and deal with since they often do not have precise definitions, and do not have clear-cut criteria of when they have been satisfied. For example, it is difficult to know how to specify the extensibility requirements of a system, and when a system has met that requirement. In addition NFRs often contradict each other — providing for extensibility (say, through a layered system architecture) may adversely affect the performance of the system.

A URN that addresses NFRs should therefore address such requirements up-front during analysis, and allow such NFRs to be expressed even if they are ill defined and tentative. The URN should then support further refinement and clarification of these ill-defined and tentative NFRs and, if possible, allow their quantification to be expressed. It should allow exposing and modelling conflicts among NFRs during analysis, and provide abilities to evaluate trade-offs among conflicting requirements, and expose and facilitate negotiation between the involved stakeholders. A URN should allow relating NFRs to potential alternative elements within the functional requirements specification such that NFRs can be used as selection criteria among them.

Sometimes the precise meaning and the degree of achievement of NFRs may become clear only during the design stage, or even during implementation of the software system. A URN should therefore also support and guide the process of

refining and clarifying NFRs to serve as selection criteria during these later phases of the software lifecycle. This requirement for a URN emphasises the need for linking NFRs to all phases of the software development life cycle, since the degree of achievement of NFRs may be affected by decisions during all phases. A URN should provide support for managing, tracing, validating and evolving NFRs, as well as functional requirements (FRs), during the whole development life cycle.

In order to provide for such a comprehensive life-cycle support for NFRs two requirements for a URN are introduced:

- To explicitly support goal-based modeling and reasoning for both functional and non-functional requirements as a means for relating higher-level business goals and organizational objectives to possibly alternative specification of the functional and informational (and to architectural, design and implementation) aspects of the intended software system.
- Since goals that express NFRs are initially often ill defined, tentative and ambiguous the URN should provide support for the process of refining and clarifying such goals to more precise concepts. This process support should be part of an engineering process for requirements, which acknowledges that establishing requirements is a decision making process with many interrelated activities, and which has relevance during the whole life cycle of a software system.

**Why Goal-Oriented Requirement Engineering.** Broadly speaking, many user requirements are often first stated as desired goals that stakeholders wish to achieve. Providing for a immediate way of expressing such goals, rather than activities and entities that come to achieving such goals allows reasoning and considering alternative ways stakeholder goals may be achieved by the software system. Goal-based modelling can therefore be used for the treatment of NFRs as well as the development of functional requirements. Goals that denote functionality enable expressing and reasoning about functional alternatives for which clear criteria exist and allow evaluating whether a software system in fact provides for either of the desired functionality. Goals that denote NFRs enable expressing a softer notion of achievement. It is said that an NFR related goal is *satisficed* when there is sufficient positive and little negative evidence for their achievement, and that they are unsatisficable when there is sufficient negative evidence and little positive support for their satisfiability. Unlike functional goals where automatic reasoning can (to some extent) establish whether they were fully achieved or not, NFR-related goals may need humans to intervene in cases when only weak or conflicting evidence is provided. This calls for evaluation and (achievement) decision-making that need to be made interactively (semi-automatic) by the stakeholders during the requirements analysis but also during the design and implementation process. During this analysis process both goals denoting functionality and NFRs are stated up-front, and refined to produce goal graphs. During this process the NFR-related goals are used as selection criteria among alternative functional requirements that achieve the functional goals of the stakeholders in general and of the system in particular. During design and implementation goal graphs are further refined and linked to architectural, detailed design and implementation functions and structures. Functional related goals provide for focal points of alternative design and implementation choices, while NFR-related goals provide for the selection criteria that are considered for each potential functional alternative. Design choice is made by selecting one branch for further refinement.

A URN that provides for goal modelling allows linking the elements of the software system requirement specification to their rationales, which are to be found in the system's environment. This allows capturing "why" elements of the intended specifications were proposed and reasoning whether the proposed specification is sufficient for achieving the higher-level objectives of the system and the organisation. Using goals also allows guiding the requirement process in exploring and evaluating alternative system specification, exposing conflicting interests among relevant stakeholders, and aiding the management and the evolution of requirements, when objectives change over time. Including support for the process of requirements engineering in the URN, allows reasoning about high-level objectives while they are still informal and in need for clarification, and provides support for refining those objectives towards a more precise specification. During this refinement process, alternatives may be expressed, evaluated, justified or rejected both in terms of NFRs and in terms of pertinent domain knowledge, until the requirement engineer arrives at a satisfactory specification.

### 2.1.2 Requirements for URN-NFR

This section summarises the requirements for a URN that deals with NFRs.

#### 2.1.2.1 Expressing tentative, ill-defined and ambiguous requirements

A URN that deals with NFRs needs to provide the ability to express tentative and ill-defined requirements, that are difficult if not impossible to formalise, and where there do not exist clear criteria for their achievement during requirements analysis, but also during design and implementation. Expressing such kind of requirements is in particular important during the early phases of requirements elicitation, when the understanding stakeholders have of their objectives is still vague, tentative, ill defined and ambiguous, and in need to be clarified.

#### 2.1.2.2 Clarifying and "satisficing" tentative, ill-defined and ambiguous requirements, and exploration of alternatives

A URN should provide support for clarifying and "disambiguating" such requirements in a systematic manner, through refinements, during requirements elicitation and analysis. It should provide support for exploring alternative meanings for such requirements. In addition since no clear-cut criteria exist for when such requirements are achieved, there is a need to provide a more flexible, and fine-grained notion of achievement, such as sufficiently achieved, some contribution towards achievement, some negative evidence against achievement, and insufficiently achieved. Alternative solutions would achieve such requirements with different degrees of satisfaction. Interactive, semi-automatic (i.e. not completely automated) analysis facilities, that "know" when to refer back to the analyst for her subjective opinion during evaluation, are needed to provided support for evaluating how well solutions satisfy such requirements.

#### 2.1.2.3 Support for expressing and evaluating measurable NFRs

The URN should support expressing NFRs that do have clear metrics and measurements for their achievement, and incorporate such NFRs in the reasoning and evaluation process. A particular benefit of providing support for both qualitative and quantitative NFRs is the ability to show how one is traded off for the other. Key issues in many systems include performance requirements that need support for their evaluation, together with the ability to document how and why they are traded off for other desired quality requirements of the system.

#### 2.1.2.4 Argumentation support

A URN that provides support for iterative clarification of requirements concepts should support the recording of arguments for or against such clarifications, during the refinement process. Such arguments should then be taken into account when evaluating solutions for their degree of how well they achieve requirements.

#### 2.1.2.5 Linking high-level business goals to system requirements

Since the tentative and ill-defined NFRs are often high-level organisational and system objectives, a URN should support linking such high-level concepts to the more concrete elements of the requirements specification. Such links may then provide an understanding of how intended software systems in fact contribute to the high-level, and to some extent strategic directions, an organisation wishes to take.

#### 2.1.2.6 Multiple stakeholders, conflict resolution and negotiation support

Since requirements may originate from multiple stakeholders, a URN should be able to express from where requirements originate, and whether stakeholders' interests conflict with each other.

#### 2.1.2.7 Prioritising requirements

A URN should also support prioritising requirements in general and for stakeholders in particular. This would support the negotiation process, when conflicting requirements arise, but also allows expressing the importance of requirements and how they might change over time, and in what way this may change the focal point of the system development effort.

#### 2.1.2.8 Requirements creep and churn and other evolutionary forces

A URN needs to support the ability to detect evolution in requirements between the time they are formulated and the time the product is delivered, in particular when requirements are added or changed (requirement creep). It should also support frequent modification of the same requirements or their priorities (requirement churn). Both have an impact on the requirements specification, and how changes in the requirements specification affect the rest of the development process.

#### 2.1.2.9 Support for integrated treatment of functional and non-functional requirements

A URN should enable dealing with both functional and non-functional requirements concurrently. In particular a URN needs to express in what way NFRs may serve as selection criteria when choosing among alternative functional requirements, and for expressing constraints when wishing to achieve functional requirements during design. The designer of the URN may choose among different degrees of coupling between the elements within the URN that deal with NFRs and the ones that deal with FRs.

#### 2.1.2.10 Support for multiple "layers" of commitment

Moving from high-level objectives to software system requirements may need multiple rounds of decision-making and commitment by the relevant stakeholders. Each new round of decision making is based on previously adopted decisions, that structures the potential "decision space" in particular ways — by focusing on certain "strategic" alternatives, and excluding others. In addition NFRs need to be "re-addressed" whenever requirements specification, design and/or implementation elements, are introduced. Performance issues, for example, exist during the whole development

lifecycle, even when particular decisions are adopted for dealing with performance. A URN should therefore support for multiple rounds or layers of decision-making, where each layer proceeds from commitment points of previous layers.

#### 2.1.2.11 Life-cycle support

Since requirements and their management are relevant during all phases of software development a URN should support the dealing with requirements during all phases. This includes the exploration during early requirements, where principle system-and-environment alternatives are explored, requirements specification, where organisational objectives are refined into precise specifications of the intended software systems, and the linking of requirements to high-level and detailed design, implementation and testing.

#### 2.1.2.12 Forward and backward engineering and traceability support

The URN allows expressing NFRs as ill defined, and tentative during early requirements, and provides 'process' support for refining those requirements to a more precise specification. It also supports using NFRs to guide the decision making process during design and implementation. This 'forward engineering' provided by the URN supports traceability from requirement to design and to implementation. This conforms to the definition of traceability given in which is the ability to describe and follow the life of a requirement in both forward and backward direction (i.e. from its origins, through its development and specification, to its subsequent deployment and sue, and through all periods of on-going refinement and iterations in any of these phases).

#### 2.1.2.13 Ease of use and precision

A URN is used by many different stakeholders during the requirements analysis and development process. For some stakeholders ease of use, and comprehensibility, while for others, precision in expressing requirements, is of great importance. A URN should provide support for both types of URN users, through supporting degrees of formality in its language, and by making clear how a user of a URN would refine from rather informal to formal expressions of requirements. One focal point should be ease of use for practitioners, and comprehensibility for customers and intended users of the system, while another focal point should be the ability to specify requirements more precisely for stakeholders in the development process.

#### 2.1.2.14 Knowledge-based support (catalogues of reusable requirements and design knowledge)

A URN should allow reusing parts of requirements specification, which comes to achieve certain known objectives, when such or similar objectives recur in other projects. Such a URN that supports a knowledge-based approach to requirements analysis would provide facilities to capture, structure and reuse knowledge, and know-how related to developing a requirement specification. Know-how of achieving functional and, in particular, non-functional objectives could then be captured and distilled within knowledge catalogues, and reused when similar such requirements (or design situation) arise. Such catalogues could also capture design and implementation know-how that come to achieve functional and non-functional requirements of the software system. Knowledge would be captured together with applicability conditions stating under what circumstances it is intended to be reapplied. In addition, the URN should allow for arguing for or against the use of knowledge, when particular application depended factors, further encourage or invalidate the use of that knowledge. Such knowledge can then be utilised to provide guidance, and point out trade-offs, during analysis, design and implementation.

### 2.1.3 What is a URN-FR?

An URN-FR model is an abstract representation of the behaviour of a proposed system and its environment. The stakeholder can use the URN-FR to specify scenarios, that is, sequences of responsibilities that must be executed to transform input events and preconditions to output events and post-conditions. Scenarios are also called maps because they look like roadmaps connecting inputs to outputs. The notation also allows the user to specify relationships among scenarios. It is possible to use the notation to specify components and allocate responsibilities to them but it is not necessary to do so. Responsibilities are connected by causality flows. A causality flow is the assertion of a causal connection between responsibilities. That is, the execution of this sequence of responsibilities in some fashion causes or enables the execution of a subsequent responsibility. The intent behind the notation is to leave the specification of detailed interactions between responsibilities to more concrete notations such as Message Sequence Charts. The goal is to allow stakeholders to express their domain knowledge in an intelligible way without letting detailed design considerations get in the way.

A URN-FR must facilitate negotiations between stakeholders and implementers. The aim is to use the URN-FR model to discover as many policy questions as possible so that stakeholders can rule on these questions prior to implementation. The URN-FR notation can be used for high-level design as well as for functional requirements specification. For the latter purpose a subset of the syntactical elements is used, and the specification is less refined than that of a high level design, particularly with respect to architectural definition. Developers can begin the high-level design phase by iterating the model constructed by the stakeholders. If, by doing so, the developers discover requirements, they can discuss the matter with the stakeholders by referring to the model.

The initial version of the URN-FR is evaluated for clarity, consistency, correctness, and completeness by visual inspection. In other words, there is no formally defined algorithm for validating a URN-FR model. Evolving the URN-FR so that it can be validated is a goal. When this goal is achieved, this standard shall contain a specification for the validation algorithm.

Before listing the URN-FR capabilities, it is necessary to define some terms. *System* means new system or system extension. A *scenario* is the set of ordered responsibilities a system must perform to transform inputs to outputs and preconditions to post-conditions. A *feature interaction* is the set of conditions under which the execution of one scenario is affected by the execution of another. *Textual information* is information stakeholders believe is important for designers and implementers to know but is not considered when validating the functional requirements model. *Preconditions* are the relevant set of values in the system data when an input event occurs. *Post-conditions* are the relevant set of values in the system data when a scenario completes and has generated the set of output events.

### 2.1.4 Requirements for URN-FR

This section summarises the requirements for a URN that deals with FRs.

The Functional Requirements URN (URN-FR) must allow stakeholders to:

*Specify trigger and termination conditions for system responses*

- Specify the set of input events at a scenario start point
- Specify the set of output events at a scenario end point
- Specify preconditions at scenario start points
- Specify post-conditions at scenario end points
- Identify input sources, that is, whether the sources are human or machine
- Identify output sources, that is, whether the sources are human or machine

*Specify what the system does in response to trigger conditions*

- Specify system operations in terms of a causal flow of responsibilities
- Specify alternative courses of action within a scenario
- Specify repetitive action within a scenario
- Specify parallel courses of action within a scenario
- Specify synchronization within a scenario
- Specify synchronization between scenarios

*Specify lengthy system responses in a hierarchical way*

- Specify a lengthy scenario by way of a root map and references to child maps; child maps may have children
- Specify preconditions at the entry points to a child map
- Specify post-conditions at the exit points from a child map

*Specify relationships among scenarios*

- Group related scenarios
- Specify feature interactions

*Specify component architectures, if appropriate*

- Specify scenarios without reference to components
- Specify scenarios with reference to components and the allocation of responsibilities to components
- Specify scenarios with reference to Commercial-Off-The-Shelf (COTS) components
- Specify scenarios with reference to conceptual components

*In general*

- Specify the behaviour of the system’s environment

- Elicit requirements, that is, use the notation to reason about domain knowledge
- Cross-reference operationalizations in the NFR model to responsibilities in the FR model
- Cross-reference performance constraints identified in the NFR model to responsibilities or scenarios in the FR model

#### 2.1.4.1 Specifying system trigger and termination conditions

A requirements specification, if it contains nothing else, must contain a mapping of input events and preconditions to output events and post-conditions. Preconditions and post-conditions relate to both environmental states and target system states. The environmental set of preconditions and post-conditions are kept separate from the system set by the fact that one set of scenarios models the environment and one set models the system. The URN-FR is used to model both the environment and the target system. The start points of the system scenarios are connected to the endpoints of scenarios occurring in the environment, and the endpoints of the system scenarios are connected to the start points of scenarios occurring in the environment.

The URN-FR must allow the client to distinguish the many mappings of input events and preconditions to output events and post-conditions for a particular system in whatever degree of detail seems appropriate. It is not sufficient just to be able to identify an event class or condition parameter; it must also be possible to specify ranges of values in each class or parameter. A large number of classes, parameters and value ranges create the possibility of many mappings; data management becomes an issue.

A goal for this standard is to define a data model so that preconditions, input events, post-conditions and output events can be formally defined and managed.

#### 2.1.4.2 Specifying system responses

This notation allows users to specify system responses as a causal flow of responsibilities. The execution of a responsibility is said to cause the execution of a subsequent responsibility. Inter-responsibility communication is not specified.

A system response is what the system does to transform the input events and preconditions into output events and post-conditions. Given the many possible mappings between input events and preconditions on the one hand and output events and post-conditions on the other, how to manage the specification of system responses for each of these mappings becomes an issue.

One candidate solution is to group scenarios according to event classes. The event classification is based on common processes and criteria of relatedness. An example of an event class is that of bit patterns on a receive link where a synchronous data protocol is being used. The system response may differ somewhat based on which event in the class is received. To express this difference, notation for condition-based decision-making (branching) must be used. The handling of preconditions also requires branching. Preconditions express system state. For example, the system may be in an operational state relative to a particular event class when it receives the event or it may be out of service. Branching is used to express the different response the system makes depending on its state. Branching is also called OR-forking. A goal for this standard is to define a data model and expression evaluator so that conditions on OR-forks can be formally expressed.

Another name for a scenario specification is a map because in its graphical form it looks like a map.

Events in the same class may be handled in much the same except for slight differences. The map must show where the common processing segments are as well as where the branching segments are. It's possible that after a branch, the system handling for 2 events may again be the same for a while. The notation must be capable of expressing this situation and does so using an OR-join.

The notation must be able to express parallelism when specifying the handling of an event. For example, the detection of a loss of signal on a receive link causes two parallel actions to be taken. The first action is to send an alarm out on the transmit link to the far end, and the second action is to send an alarm to the human user interface.

The notation must be able to express synchronization when specifying the handling of an event. For example, some bank vaults can only be opened when two people physically out of touch have inserted and turned their keys. The system waits until both events have occurred before proceeding. The notation must be able to express a wait-forever condition as well as a timed wait with action attendant on a timeout.

The notation must be able to specify repetitive action. Collecting digits during a call-set-up is a classic example of a repetitive action that can be expressed as a loop. A goal for this standard is to define a data model and expression evaluator so that conditions on loops can be formally expressed.

#### 2.1.4.3 Specifying lengthy behaviour

The notation must allow the user to specify lengthy system responses in a comprehensible way. One way to support comprehensibility is to support abstraction, that is, hide irrelevant detail. In a requirements specification some detail is always irrelevant, for example, how responsibilities communicate with each other. Another way is to support hierarchical decomposition of the scenario specifications.

The mechanism used for hierarchical decomposition is the stub and plug-in notation. The stub symbol replaces a sequence of responsibilities in a higher-level map. The replaced sequences are represented in a lower level map. This separate specification is called a plug-in. A plug-in is similar in appearance to a scenario since it has a trigger symbol and termination symbols. The term map can also be used to refer to a plug-in. A stub is static if only one plug-in is defined and is dynamic if more than one plug-in is defined. In the latter case a selection policy (related preconditions) determines which of the alternate plug-in executes.

The stub and plug-in mechanism allows clients to specify lengthy behaviour in a layered way with details left to lower level maps. This nesting of maps can be arbitrarily deep. On the one hand the ability to layer the maps aids in comprehension since it prevents any one map from becoming too cluttered. On the other hand, it is possible that too much layering could impede comprehension. See section 8 for more discussion of this issue.

Stubs and plug-ins can be used to encapsulate behaviour that is found in many places within one scenario or across scenarios. It is a behavioural component.

#### 2.1.4.4 Specify relationships among scenarios

One form of relationships is grouping a set of scenarios that deal with a class of events into a single specification.

Another form of relationship is synchronization. Consider the example of a call set-up. The system detects an off-hook, responds with dial tone, sets a timer and waits. This is a single scenario. The system has received an event and produced outputs. Subsequently the system detects a digit, removes dial tone, resets a timer and waits. This also is a single scenario, but it is related to the first because it causes the timer in the first scenario to be reset.

The URN-FR must allow the user to express desirable feature interactions and discover undesirable ones. For example, under certain conditions a particular service may receive priority treatment, causing the interruption and delay of a lower priority service under way.

#### 2.1.4.5 Component definition

The URN-FR must allow the user to specify scenarios without reference to components as well as with reference to them. The URN-FR can thus be used in situations where no component architecture has yet been defined and where there is a desire to put no architectural constraints on implementers, and it can also be used where a component architecture has been defined.

Component definition internal to the system is more appropriate to high-level design than to requirements specification because it involves allocation of responsibilities to components. Allocation of responsibilities to components is a high-level design activity, and many criteria are applied to determine a good architecture. Nevertheless, stakeholders may feel more comfortable if they can reference entities in the specification. These entities should be considered abstract, functional entities and not instructions to implementers on responsibility allocation unless the entities are COTS components.

Component definition is appropriate when the system environment is specified in terms of existing components, and the functional model encompasses both the existing components and the new system.

In general, the functional model will focus on behaviour, and component definition will be left to the high-level design phase.

#### 2.1.4.6 Specifying the environment

The clients must be able to specify the behaviour of the system's environment. All of the capabilities of the URN-FR that can be used to model the system can be used to model the environment. The environment model then becomes the driver for the system model and vice versa.

Component definition can come into play here. The system is identified as a single component and is connected to existing systems modelled as black box components in the new system's environment. The value in this level of component definition is that it clarifies what behaviour in the overall scenario specification belongs to the new system and what belongs to the system's environment.

### 2.2 Relationships between the URN-NFR and URN-FR models

The URN-NFR modelling exercise sets the context for the URN-FR modelling exercise.

Any operationalizations and associated performance constraints identified in the URN-NFR model must be cross-referenced to the responsibilities that represent them in the URN-FR model and vice versa.

## 2.3 Requirements traceability

In a software engineering process, *traceability* is the property that defines how elements contained in different system models relate to each other. This allows linking model elements that are semantically related.

In the specific context of the URN, requirements traceability is of particular importance. *Requirements traceability* is the property that allows linking system artefacts defined in the different models and design decisions to requirements.

In a software development process, the definition of requirement traceability relations is important for many reasons:

- **To evaluate requirements coverage**. An important question that a software engineer must be able to answer is: Are all requirements addressed in the current version of the system? In order to answer this question, one must be able to determine precisely the set of requirements that are referenced in the different system models. If requirements traceability relations have been maintained during the whole design process, this question can be easily answered. Moreover, the set of requirements that have not yet been addressed can then be automatically determined.

- **To evaluate the impact of requirements modifications**. Another important question that a software engineer must be able to answer is: What are the model elements that are related to a specific requirement? This question must often be answered in the case where modifications are made to requirements. The existence of traceability relationships allows evaluating the impact of modifications on the different models, and making the changes to affected models in a consistent manner. Thus, if a modification is made to a requirement, say R1, designers can evaluate the impact of the modification by analyzing the elements of the different models that are linked to R1.

- **To allow requirements testing.** In a scenario-driven (or use case-driven) process, requirements are associated with specific scenarios. Therefore, in order to test that the current implementation of a system is correct with respect to a specific requirement, one must first determine the set of scenarios that are related to the requirement. Then, the set of scenarios can be executed, and the result of the execution can be analyzed to see if the requirement is correctly addressed or not. For this purpose, it is important to establish traceability relations between elements of the scenario descriptions in the URN and stakeholder requirements.

- **To allow the identification of conflicting requirements**. The causes of system errors are various. One important cause of errors is conflicting requirements. This type of error is often difficult to prevent and is only discovered late in the development process. For this reason, when an error is found in the system, it is important to be able to trace it back to the different models, and ultimately to requirements, and see where the error has been introduced. If the error comes from conflicting requirements, then these requirements can be precisely identified.

- **To reduce maintenance efforts**. An important part of the cost of system maintenance is related to the evaluation (or the non-evaluation) of the impact of modifications. If one can determine precisely the set of model elements that can be impacted by the modification of a specific requirement (or model element), then the cost of modification would be significantly reduced.

- **To preserve the rationale for design decisions**. Knowing the original reasons for design decisions helps maintainers and enhancers to evaluate whether implementation should be changed in the light of new circumstances. Such re-engineering of implementations may be essential to keeping a product vital and competitive in the market-place. The URN-NFR notation should provide the ability to present the rationales for a specific choice together with the arguments for it, in a concise and readable form.

In the context of the URN, we need to define both *backward traceability* relations from the URN, and more specifically URN elements, to their source (documents, stakeholders requirements, problem domain analysis, etc), and *forward traceability* relations from the URN to the other models used in the process in which the URN is used (or preferably backward traceability relations from the other models to the URN). If traceability exists between the other models and implementation, the existence of these two types of traceability relations would transitively ensure a complete traceability from implementation to the source of requirements.

## 2.4 Requirements test case specification

URN should support the testing of requirements as well as testing based on requirements. A requirements test case specification describes scenarios found or expected to be found in the URN-FR specification. The URN-FR specification is assumed to include operationalization of relevant non-functional requirements; hence part of the URN-NFR specification (e.g. quantitative performance attributes) is indirectly tested at the same time. The requirements test case specification aims to enable the following types of testing:

- *Validation testing*, used to capture individual or small-grained client and user scenarios so that the integrated set of requirements can be determined to be valid by the clients and users. This type of testing can be used by clients and developers to establish contract satisfaction.
- *Conformance testing*, used to verify designs and implementations against the requirements. Such test cases could be created in way that would improve compatibility with the ITU-T testing language, TTCN-3.
- *Regression testing*, used at the requirements level to ensure a certain degree of compatibility with key system properties during the evolution of requirements.
- *Dynamic assessment* is used in dynamic systems that need to assess capabilities of components and other systems with which they communicate (for instance, does this unknown component supports this quality of service?). This dynamic assessment may involve testing of the component or other system in question. URN is required to support the means of describing such dynamic assessment tests.

The testing of non-functional requirements in general is also desirable but may not be achievable through the use of scenarios. The URN notation may not be able to support this type of testing.

## 2.5 Performance analysis of functional requirements

URN should support at least a preliminary analysis of performance properties, such as response delay or throughput capacity, based on workload and environment parameter estimates attached to the URN-FR specification. Performance properties are of critical importance in telecommunications, and current work indicates that the analysis is feasible. The necessary workload parameters include

- *Scenario triggering parameters* such as period of initiation, distribution of delays between initiations, etc.
- *Frequencies* of alternative paths.
- *Processing demands* of scenarios, and of operations within scenarios.
- *Demands for system services* other than processing, made by scenarios and operations.

The integration of these parameter types into URN-FR is described in Section 5.2.13. The environment parameters should approximately describe the processing capacity, network delays and the services provided by the environment (for instance, a response delay for a remote service).

Performance results will be delays along defined processing paths, or the range of possible throughputs of some scenarios. The intention of the analysis is to estimate the degree of conformance with stated performance requirements, and to identify problem areas and sensitivities. The data and the results are expected to be approximate. The analysis can be performed in various ways:

- *Point analysis* considers one set of conditions.
- *Sensitivity analysis* considers a range of conditions, and the variation of performance measures with parameter values. This could include sensitivity of the system to its workload parameters or to the environment.

## 2.6 Legal status of URN models

Functional requirements constitute an essential part of a contract between stakeholders, for example, clients and implementers. The URN-FR must be usable in a legal setting.

## 2.7 Change management

The URN notation has textual and graphical elements. It must be possible to version control URN models.

## 2.8 Intended usage

### 2.8.1 Communicating requirements

A primary purpose of a URN is to facilitate the communicating of requirements among pertinent stakeholders prior and during the software development life cycle in general and during requirements analysis in particular. This includes communicating among stakeholders such as clients, standards bodies, business analysts, intended users of the system, architects, designers, implementers and the like. If we take standards bodies as example, it will show the significance of utilizing URN for these bodies. Industry standards are dynamic in nature, continuously evolving to meet stakeholders' requirements with ever-shorter intervals for standards development. The current timelines at which a new version of the specification is to be completed to the needed level of precision, quality and completeness cannot be accommodated using existing specification techniques. A key assumption is that future standards work must apply techniques that can be

automated. The use of formal documentation techniques using tools will shorten the standards development cycle, introduce a formal test methodology, and assist in rapid validation and verification, harmonization, and evolution of the standards.

In addition during the early phases of requirements elicitation (and high-level design) a URN should support an exploratory mode of work, in which principal alternatives are considered and where minute details are omitted and left for future elaboration. Such exploratory work is often done in conjunction with non-technical stakeholders, in order to explore feasible directions towards system specifications (and design). Having achieved agreement on principal directions the URN should support detailed specifications that may then be undertaken in conjunction with more technical oriented stakeholders, and that allow for validating of requirements in a formal manner.

These considerations give rise to three requirements on a URN:

- The ability to provide for informal (or semi-formal) requirements descriptions that focus on coarse grained behaviors and structures of the intended system. Such a description would facilitate the exploring of alternatives and would omit details not pertinent to such reasoning. This would facilitate the communicating of requirements among non-technical oriented stakeholders.
- The ability to provide for formal requirements descriptions that do focus on detailed behaviors and structures, such as system state and state transition information and how and where these are represented and changed within the system structure. This would facilitate the communicating of requirements among technical oriented stakeholders.
- The ability to support the transition from informal (or semi-formal) descriptions to formal ones, together with the ability to reason about and explore alternative "formalizations" during that transition. This would provide the basis for communicating among both non-technical and technical oriented stakeholders.

This URN proposal acknowledges the abilities of existing requirements notations (such as UML, SDL, MSC) but positions their abilities as belonging to the more formal and detailed requirements description approaches. This URN proposal wishes to address those earlier phases during requirements analysis when a more exploratory, coarse-grained and informal approach is more suitable, that does not overwhelm the stakeholders with irrelevant details and provides support for exploring alternatives.

These early phases of requirements analysis are not well supported by existing requirements analysis notations. This URN then supports the transition from such informal descriptions to formal ones, and will to a certain extent support in the future the specification of more detailed requirements specifications in terms of state and state transition behaviour.

In order to further provide aids for communicating requirements, this URN proposal supports graphical, textual and tools-oriented representation of requirements. The graphical representation allows the modelling of requirements in an iconic and spatial manner, which facilitates an intuitive understanding and emphasizes the informal aspect of the URN. The textual representation provides a language rendering of the graphical notation but provides further expressive capabilities not available in the graphical notation. These additional language facilities could be made accessible either through addition user interface elements, such as pop-up dialogues and context dependent menu items, or could be written and parsed from textual language representation. The textual representation is provided for more technical-oriented stakeholders who might prefer such representations to graphical ones. Finally, a tool-oriented representation provides for the ability to exchange requirement data between different tools in a standardized manner.

### 2.8.2 How architectural constraints affect requirements specification

#### 2.8.2.1 New system with no architectural constraints

There is no sub-system architecture. The only components identified are the system as a black box and existing systems in the new system’s environment to which the new system is connected. The scenario specifications are purely behavioural. There may be constraints on responsibilities where the URN-NFR model has specified a particular technology option to operationalize a quality attribute. For example, to operationalize the security attribute the URN-NFR specifies that the user has to enter a userid and password.

#### 2.8.2.2 New system based on COTS components

The stakeholders have identified Commercial-Off-The-Shelf (COTS) components to be used to implement the operationalizations specified in the URN-NFR model. Stakeholders specify scenarios in terms of these components by drawing the causality flow line through responsibilities in these components. Where the scenario requires a responsibility that is not implemented in one of the COTS components, the stakeholder draws the causality flow outside the components and places the notation for the responsibility there.

At the requirements level there is no need to specify responsibilities whose purpose is to glue COTS components together.

#### 2.8.2.3 Extension of an existing system

This case is similar to the case of a new system based on COTS components in that both specifications involve the use of existing sub-system components. If the existing system has been modelled using the URN from its inception, and if the URN-FR and URN-NFR models are up-to-date, then the approach used in the new system specification based on COTS components is applicable here. If, however, there is no URN-FR model of the existing system, the stakeholders must first construct one. How to reverse engineer an URN-FR model from an existing system is not within the scope of this standard; it may, however, be an appropriate subject for a related standard.

An option may be to treat the specification of the extension as a new system with no architectural constraints. It will then be up to the implementers to map the responsibilities identified in the scenario specifications to capabilities in sub-components in the existing system.

### 2.8.3 Refining URN-FR models

The URN-FR originated because of a need to specify high-level design in a way that highlights the important semantics but does not let details about implementation mechanisms get in the way of reasoning about what should happen. When starting out a specification, the user is expected to use the notation in an informal manner. The intent is that the notation be abstract and let the user stay at the level of the big picture. An important attribute of the notation is that it allows the user to create artefacts that are meaningful even when the process of thinking about requirements is in its initial phases. Informality is marked by a focus on defining high-level responsibilities, causal relationships between responsibilities, and control flow. The user is allowed to focus on one scenario at a time; this greatly simplifies the thinking process. Although the mapping of input to output events is there, detailed specifications of data transformations internal to the scenario is missing. The current absence of a data model in the URN-FR ensures that detailed specifications of data transformations do not occur.

The addition of a data model to the notation will mean that further refinement of the URN-FR model can happen. It also means introducing more formality into the specification since a data model is a prerequisite to executability and validatability. The user will have to deal with more detail but it is not detail about implementation mechanisms. Rather it is detail that relates to cases that the proposed system must handle and is therefore germane to a requirements specification. Tool support based on formal semantics will help the user manage the complexity that inevitably arises within specifications for products, protocols, and features that will be integrated into today's telecommunication networks. The focus is still on scenarios and the direct mapping of inputs to outputs. The user is not being asked to architect components. As well the data definition can be at a much higher level of abstraction than is required for an implementation.

## 2.9 Quality attributes

This section discusses the quality attributes that are applicable to the URN and any implementation of it.

### 2.9.1 Usability

One quality attribute that the URN and any implementation of it in a tool should have is usability. Usability is key to user acceptance, and user acceptance validates the standard-making process and motivates its continuance. How to test a URN implementation for usability is an issue for further study.

## 2.10 Round trip engineering

For the purpose of this standard round trip engineering can be said to take place when either of the following two situations obtain. Users of a tool implementing the URN suggest changes to the tool, and by some form of usability testing these changes are determined to be improvements. Users of the URN suggest changes to the notation, and by some form of usability testing these changes are determined to be improvements.

# 3 URN language specification

The URN language is composed of two related and complementary parts: the URN-NFR language (Section 4) and the URN-FR language (Section 5). The current section provides the basis for integrating these two languages with an XML document type definition.

## 3.1 XML DTD for the User Requirements Notation

The URN language is a graphical notation that supports annotations. The language also has a textual representation described in XML (W3C's eXtensible Markup Language) in a URN Document Type Definition (DTD). XML DTDs describe the syntax of languages in terms of elements and their attributes. Elements define the document structure by

describing containment rules, and attributes describe mandatory and optional variables and their data types for further qualifying elements and references to elements.

XML is an appropriate language for describing the integration of the URN-NFR language with the URN-NFR language. Also, XML supports a simple evolution path towards the integration of URN with UML through technologies such as OMG's Meta-Object Facility (MOF) and XML Metadata Interchange (XMI).

The remaining sections will present the DTD element per element. Note that not all URN elements will have a visual representation; many elements are in fact structural elements that contain other elements, or textual annotations attached to other elements. A graphical representation of the whole DTD can be found in Annex A. Annex A also contains basic information on how to read XML DTDs.

## 3.2 Conventions

URN DTD elements and attributes will be described inside shaded boxes using the `Courier` font. XML elements use lowercase characters (`element`) whereas XML attributes are in lowercase-italic (*`attribute`*) and XML keywords are in uppercase-bold (**`KEYWORD`**).

In the URN-FR language specification, thin arrows are sometimes used to indicate the direction of scenario paths and are not part of the URN-FR notation. Three dots (…) are used to indicate incomplete scenario paths but without suggesting any direction. These three dots are not part of the URN-FR notation either.

Graphical notation elements do not have figure numbers, and the related explanations are found below the figures.

## 3.3 Structure of URN specifications

The URN language supports the description of URN specifications composed of a specification of goals and non-functional requirements (`urn-nfr-spec`) and of a specification of functional requirements (`urn-fr-spec`). It also supports global `definitions` and additional `annotations`.

In the DTD, the element `urn-spec` is defined as the *root element*, and it contains an identifier (*`spec-id`*), a name (*`spec-name`*), and the version of the DTD being used (*`dtd-version`*). The *`component-notation`* attribute indicates the component notation used in the URN-FR specification. The default notation is Buhr's component notation, but eventually other notations may be supported (e.g. UML, UML-RT, SDL structures, or architecture description languages such as ACME and Wright). A URN specification can also make use of a *`data-language`* (e.g. ASN.1, Abstract Data Types, etc.), but none is being suggested or imposed at the moment. The size of the drawing workspace (*`width`* and *`height`*) can also be specified in terms of pixels. The `urn-spec` element, like most elements in this DTD contains a free-text *`description`* attribute.

The definitions of components, responsibilities, and non-functional requirements are covered in upcoming sections.

#### 3.3.1.1 XML definition

```
<!ELEMENT urn-spec (definitions?, urn-nfr-spec?, urn-fr-spec?,
                    annotations?)>
<!ATTLIST urn-spec
                    spec-id             ID      #REQUIRED
                    dtd-version         NMTOKEN #REQUIRED
                    spec-name           CDATA   #IMPLIED
                    component-notation  NMTOKEN "Buhr-UCM"
                    data-language       NMTOKEN "none"
                    width               NMTOKEN "1320"
                    height              NMTOKEN "1100"
                    description         CDATA   #IMPLIED >

<!ELEMENT definitions (component-definitions?, responsibility-definitions?,
                       nfr-definitions?)>
```

#### 3.3.1.2 Graphical notation

No graphical notation is required for these DTD elements.

### 3.3.2 URN-NFR specifications

The description of non-functional requirements specifications is covered in Section 4. The NFR language in use is the Goal-oriented Requirements Language (GRL). Three kinds of representations are defined for URN-NFR. A graphical notation is intended to be used by stakeholders who need not have much technical background, and do modelling in an intuitive way. A textual grammar serves the aim of documentation, and also to stakeholders who prefer to read and write

requirement in textual style. The XML definition is mainly used as an implementation-oriented form of URN-NFR model; it is also a means to integrate URN-NFR models with URN-FR and other frameworks.

#### 3.3.3 URN-FR specifications

The description of non-functional requirements specifications is covered in Section 5. The FR language in use is the Use Case Map (UCM) notation. Two kinds of representations are defined for URN-FR. The graphical notation is intended to be used by all stakeholders. The XML definition is mainly used for formalisation and as an implementation-oriented form of URN-FR model; it is also a means to integrate URN-FR models with URN-NFR and other frameworks.

## 4 URN-NFR/GRL language specification

The URN-NFR language is based on GRL (Goal-oriented Requirement Language), which is a language for supporting goal-oriented modelling and reasoning of requirements, especially for dealing with non-functional requirements. It provides constructs for expressing various types of concepts that appear during the requirement process. There are three main categories of concepts: intentional elements, links, and actors. The intentional elements in GRL are goal, task, softgoal, and resource. 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 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.

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 kind of details during early phases of analysis (and design) allows taking a higher-level (sometimes called a strategic stance) towards modelling the current or the future 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 utilising capabilities of the software system and/or other stakeholders, by trying to rely upon and/or assign capabilities and by introducing constraint how those capabilities ought to be performed.

URN-NFR/GRL provides support for reasoning about scenarios by establishing correspondences between intentional GRL elements and non-intentional elements in 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.

### 4.1 GRL specifications

#### 4.1.1 Non-Intentional Element Definition

The language provides syntax for defining non-intentional elements. A non-intentional element may be imported from an external model. The main concern of these clauses is not to capture the syntax and semantics of the external model but to serves as references to the external model only. Non-intentional elements definition is used to navigate through the non-intentional model. The following statement defines a non-intentional element.

> **ELEMENT** <Element name> [<Informal Textual Description>] [**IS** <External Name> **FROM** <Model Name>]

##### 4.1.1.1 Textual notation

<GRL Specification> : :=

[<Non-Intentional Element Definitions>]

<Goal Model Definition>

<Non-Intentional Element Definitions>::=

< Element Definition> { < Element Definition> }$_0^n$

<Element Definition> : :=

**ELEMENT** <Element Name>

[<Informal Textual Description>]

[**IS** <External Name> **FROM** <Model Name>]

#### 4.1.1.2 XML definition

```
<!ELEMENT   nfr-definitions   (element-definitions,   device-directory?,   data-store-
directory?)>

<!ELEMENT urn-nfr-spec (grl-spec)>

<!ELEMENT grl-spec (goal-model)>

<!ELEMENT element-definitions (element)* >

<!ELEMENT element (model-name?, external-name?) >

<!ATTLIST element
              element-id          ID                  #REQUIRED
              name                CDATA               #IMPLIED
              description         CDATA               #IMPLIED>

<!ELEMENT model-name EMPTY >

<!ATTLIST model-name
              model-id            ID                  #REQUIRED
              name                CDATA               #IMPLIED
              description         CDATA               #IMPLIED>

<!ELEMENT external-name EMPTY >

<!ATTLIST external-name
              id                  ID                  #REQUIRED
              name                CDATA               #IMPLIED
              description         CDATA               #IMPLIED>
```

## 4.2 Goal model construction

A Goal-Oriented Requirement Model named <Model Name> can either be composed of a global goal model, or a series of goal models distributed in several actors.

If the model includes more than one actors, then there might exist <Dependency Series>, which represent the intentional dependency relationships between agents.

#### 4.2.1.1 Textual notation

<GRL Model Definition>::=

**GRL-MODEL** <Model Name>

<Model Constructors>

**END-GRL-MODEL**.

<Model Constructors>::= <Model Constructor> { <Model Constructor>}$_0^n$

<Model Constructor>::=

[<Actors>]

<Intentional Elements>

<Intentional Relationships>

#### 4.2.1.2 XML definition

```
<!ELEMENT goal-model (model-constructors)>

<!ATTLIST goal-model
              goal-model-id       ID              #REQUIRED
              goal-model-name     CDATA           #IMPLIED
              description         CDATA           #IMPLIED >

<!ELEMENT model-constructors (model-constructor)+ >

<!ELEMENT model-constructor (actors?, intentional-elements, intentional-relationships
) >
```

The following sections will introduce each of these kinds of constructs in turn.

## 4.3 Intentional elements

### 4.3.1 Overview

#### 4.3.1.1 Textual notation

< Intentional Elements>::= < Intentional element > { < Intentional element >}$_0^n$

< Intentional element> ::= <Task> | <Goal> | <Resource> | <Softgoal> | <Belief>

#### 4.3.1.2 XML definition

```
<!ELEMENT intentional-elements (intentional-element)+ >

<!ELEMENT intentional-element (goal | softgoal | task | resource | belief)>
```

### 4.3.2 Goal

A 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 express goals regarding the business or state of the business affairs the individual or organisation wishes to achieve. System goal expresses goals the target system should achieve, which, generally, describe the functional requirements of the target information system.

#### 4.3.2.1 Textual notation

<Goal> ::= **GOAL**<Goal Name> [<Informal Textual Description>] [ **ATTRIBUTE** <Attributes>]

[**OWNER** <Actor Name>]

<Attributes > ::= <Attribute > { <Attribute >}$_0^n$

<Attribute> ::= <Element Name>

#### 4.3.2.2 Graphical notation

<Goal> ::= <Goal Symbol>

**CONTAINS** <Goal Name > [Attributes]

<Goal Symbol> ::=

In the Graphical notation, the following keywords are used to denote specific graphical meanings:

a) **CONTAINS**: indicates that its right-hand argument should be placed within its left-hand argument.

b) **IS CONNECTED TO**… **BY**: means that its right-hand argument is connected to its left-hand argument by the kind of link type referred to by the argument after keyword “BY”.

c) **IS ASSOCIATED WITH** indicates that its right-hand argument should be positioned next to its left-hand argument.

#### 4.3.2.3 XML definition

```
<!ELEMENT goal (attributes?, actor-ref?)>

<!ATTLIST goal
              goal-id              ID                #REQUIRED
              goal-name            CDATA             #IMPLIED
              description          CDATA             #IMPLIED>

<!ELEMENT actor-ref EMPTY>

<!ATTLIST actor-ref
              actor-id-ref         IDREF             #REQUIRED>

<!ELEMENT attributes (attribute)*>

<!ELEMENT attribute EMPTY>

<!ATTLIST attribute
              element-id-ref       IDREF             #REQUIRED>
```

#### 4.3.2.4 Example

A goal is shown as rounded rectangle. For example, "Voice Connection Be Setup" is one basic goal to be achieved with any telecommunication system. How voice connection is to be set up (such as how reliable, over what medium) – may be done differently by each telecommunication provider.


In textual description, this goal would be defined with following statement:

**GOAL** VoiceConnectionBeSetup

### 4.3.3 Task

A task specifies a particular way of doing something. When a task is specified as a sub-component 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 satisfice the softgoals (called operationalizations in NFR). 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.

#### 4.3.3.1 Textual notation

<Task> ::= **TASK**<Task Name> [<Informal Textual Description>] [<Attributes>]

[**OWNER** <Actor Name>]

#### 4.3.3.2 Graphical notation

< Task > : := < Task Symbol>

**CONTAINS** < Task Name > [Attributes]

< Task Symbol> : :=

#### 4.3.3.3 XML definition

```
<!ELEMENT task (attributes?, actor-ref?)>

<!ATTLIST task
                task-id             ID              #REQUIRED
                task-name           CDATA           #IMPLIED
                description         CDATA           #IMPLIED>
```

#### 4.3.3.4 Example

A task is shown as a hexagon. Make Voice-over-LAN is one particular way of setting up a voice connection.


In textual description, this task would be defined with following statement:

**TASK** MakeVoiceConnectionOverLAN

### 4.3.4 Resource

A resource is an (physical or informational) entity, with which the main concern is whether it is available.

#### 4.3.4.1 Textual notation

<Resource> ::= **RESOURCE** <Resource Name> [<Informal Textual Description>] [<Attributes>]

[**OWNER** <Actor Name>]

#### 4.3.4.2 Graphical notation

<Resource> : :=< Resource Symbol>

**CONTAINS** < Resource Name > [Attributes]

<Resource Symbol>::=

#### 4.3.4.3 XML definition

```
<!ELEMENT resource (attributes?, actor-ref?)>

<!ATTLIST resource
                resource-id             ID              #REQUIRED
                resource-name           CDATA           #IMPLIED
                description             CDATA           #IMPLIED >
```

#### 4.3.4.4 Example

Resources are shown as rectangles. In the voice-on-the-LAN architecture, bandwidth is a resource that must be available.


In textual description, this resource would be defined with following statement:

**RESOURCE** LANBandwidth

### 4.3.5 Softgoal

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 developer to judge whether a particular state of affairs in fact achieves sufficiently the stated softgoal.

#### 4.3.5.1 Textual notation

<Softgoal> ::= **SOFTGOAL** [<Softgoal Name> **IS**]

<Softgoal Type Name> **OF** <Softgoal Topic>

[<Informal Textual Description>] [<Attributes>]

[**OWNER** <Actor Name>]

<Softgoal Topic> ::= <Element Name>

#### 4.3.5.2 Graphical notation

<Softgoal>::=< Softgoal Symbol>

**CONTAINS** <Softgoal Name> **OF** <Softgoal Topic> [Attributes]

<Softgoal Symbol> : :=

#### 4.3.5.3 XML definition

```
<!ELEMENT softgoal (attributes?, actor-ref?)>

<!ATTLIST softgoal
                softgoal-id             ID              #REQUIRED
                softgoal-name           CDATA           #IMPLIED
                description             CDATA           #IMPLIED>
```

#### 4.3.5.4 Example

A softgoal, which is "soft" in nature, is shown as an irregular curvilinear shape. For instance, reliability of router is a softgoal to be achieved during the design of a telecom system.


In textual description, this softgoal would be defined with following statement:

**SOFTGOAL** ReliabilityOFRouter

### 4.3.6 Belief

Beliefs are used to represent design rationale. Beliefs make it possible for domain characteristics to be considered and properly reflected into the decision making process, hence facilitating later review, justification and change of the system, as well as enhancing traceability.

#### 4.3.6.1 Textual notation

<Belief> ::= **BELIEF** <Belief Name> [<Informal Textual Description>] [<Attributes>]

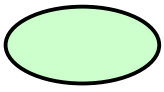
[**OWNER** <Actor Name>]

#### 4.3.6.2 Graphical notation

<Belief>::=< Belief Symbol>

**CONTAINS** < Belief Name> [Attributes]

<Belief Symbol> : :=

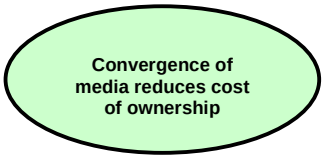


#### 4.3.6.3 XML definition

```
<!ELEMENT belief (attributes?, actor-ref?)>

<!ATTLIST belief
              belief-id              ID              #REQUIRED
              belief-name            CDATA           #IMPLIED
              description            CDATA           #IMPLIED>
```

#### 4.3.6.4 Example

A belief is shown as an ellipse. In the following is an argument supporting that the task of VoiceLAN is lowering costs.




## 4.4 Intentional Relationships

Each model structure is a link connecting two elements. These structures together constitute the overall goal model. So they are seen as the basic building blocks of models.

#### 4.4.1.1 Textual description

<Intentional Relationships>::= < Intentional relationship > { < Intentional relationship>}$_0^n$

< Intentional relationship>::= <Decomposition> | <Means-ends >

| <Contribution > | <Correlation >

| <Dependency>

#### 4.4.1.2 XML definition

```
<!ELEMENT intentional-relationships (intentional-relationship)+ >

<!ELEMENT intentional-relationship (means-ends
                           |decomposition
                           |contribution
                           |correlation
                           |dependency)>
```

### 4.4.2 Means-ends Relationship

GRL uses the MEANS-END statement to describe how goals are in fact achieved. Each task provided is an alternative means for achieving the goal. Normally, each task would have different types of impacts on softgoals, which would serve as criteria for evaluating and choosing among each task alternative.

**MEANS-END** < Means-Ends Identifier > **FROM** <Means element name> **TO** <Ends element name>

Graphically, a means-ends link connects an end node with the means node achieving it. In GRL, only goals are originally applicable to means-ends link. See Figure 1 as an example. However, for convenience, in short hand forms for a combined structure, tasks and resources could also be connected by means-ends links.

A Task Means_Ends Structure connects a task with the means (tasks) to achieve it directly, which is a short hand form of one Task Decomposition Structure and the related Goal Means_Ends Structure. See Figure2 as an example.

A Resource Means_Ends Structure connects a resource with the means (tasks) to make it available, which is a short hand form of one <softgoal Contribution Structure>.See Figure 3 as an example.

#### 4.4.2.1 Textual notation

<Means-Ends> ::= **MEANS-ENDS** [<Means-Ends Identifier>]

**FROM** <Means Element > **TO** <End Element >

<End Element> ::= <Goal Reference> | <Task Reference> | <Resource Reference>

<Resource Reference> ::= **RESOURCE** <Resource Name>

<Means Element> ::= <Task Reference>

<Task Reference> ::= **TASK** <Task Name>

<Goal Reference>::= **GOAL** <Goal Name>

<Resource Reference> ::= **RESOURCE** <Resource Name>

#### 4.4.2.2 Graphical notation

<Means-Ends> : :=

<Means Element>

{**IS CONNECTED TO**]<End Element>

**BY** < Means-Ends Link >}$_0^n$

<Means-Ends Link > : :=

#### 4.4.2.3 XML definition

```
<!ELEMENT means-ends (means-element, end-element)>

<!ATTLIST means-ends
            means-ends-id            ID                  #REQUIRED>

<!ELEMENT means-element (task-ref)>

<!ELEMENT end-element (goal-ref | task-ref | resource-ref)>
```

#### 4.4.2.4 Example

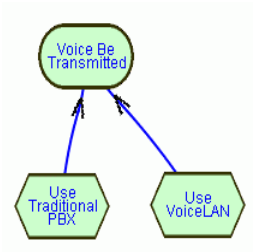



**Figure 1.** Goal means-ends structure

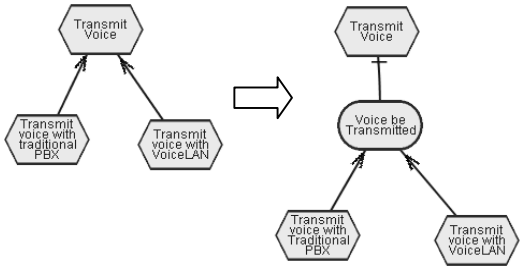



**Figure 2.** Task means-ends structure

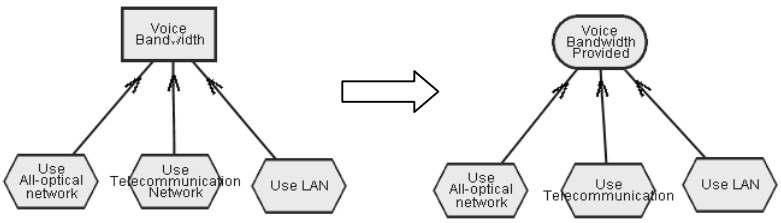



**Figure 3.** Resource means-ends structure

#### 4.4.3 Decomposition Relationship

GRL DECOMPOSITON statement provides the ability to define what other elements need to be achieved or available in order for a task to perform.

**DECOMPOSITION** <Decomposition Identifier> **FROM** <sub-element > **TO** < Decomposed Element >

Graphically, a decomposition link connects a node with its sub-components. In GRL, only tasks are originally decomposable, but for convenience, in short hand forms for a combined structure goals could also be connected by decomposition links. The sub-components of a task can be goal, task, resource, and softgoal.

A Task Decomposition Structure shows the essential components of a task, which include subtasks that must be performed, subgoals that must be achieved, resources that must be accessible, and softgoals that must be satisfied. See Figure 4 as an example.

A Goal Decomposition Structure connects a goal with its sub-goals directly, which is a short hand form of one Goal Means_Ends Structure and the related Task Decomposition Structure. See Figure 5 as an example.

##### 4.4.3.1 Textual notation

<Decomposition> ::= **DECOMPOSITION** [<Decomposition Identifier> ]

**FROM** < Sub-Element > **TO** <Decomposed Element >

<Decomposed Element> ::= <Task Reference> | <Goal Reference>

<Sub-Element> ::= <Task Reference>

| <Goal Reference>

| <Resource Reference>

| <Softgoal Reference>

##### 4.4.3.2 Graphical notation

<Decomposition> : :=

< Decomposed Element >

{**IS CONNECTED TO** < Sub-Element >

**BY** < Decomposition Link >}$_0^n$

<Decomposition Link> : :=

#### 4.4.3.3 XML definition

```
<!ELEMENT decomposition (decomposed-element, sub-element)>

<!ATTLIST decomposition
            decomposition-id        ID                  #REQUIRED>

<!ELEMENT decomposed-element (task-ref | goal-ref)>

<!ELEMENT sub-element (goal-ref | task-ref | resource-ref | softgoal-ref)>
```

#### 4.4.3.4 Example

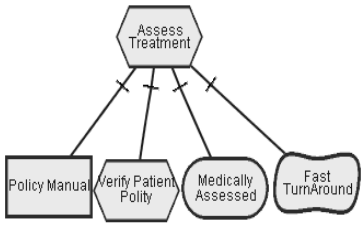



**Figure 4.** Task decomposition structure

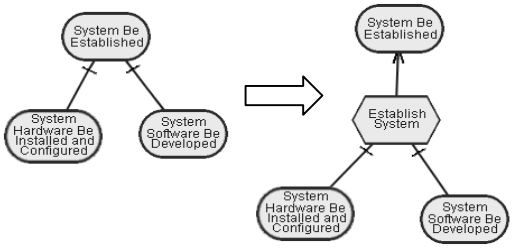



**Figure 5.** Goal decomposition structure

### 4.4.4 Contribution Relationship

The CONTRIBUTION relationship statement describes how softgoals, task, believes, or links contribute to others. A contribution is an effect that is a primary desire during modelling.

**CONTRIBUTION** [<Contribution Identifier > **IS**]

<Contributor> **HAS** <Contribution type> **CONTRIBUTION-TO** <Contributee>

Graphically, a contribution link describes how one intentional element contributes to the satisficing of another intentional element.

- *AND contribution*: The relations between the contributing elements are “AND”. Each of the sub-components is positive and necessary.
- *OR contribution*: The relations between the contributing elements are “OR”. Each of the sub-components is positive and sufficient.
- *MAKE contribution*: The contribution of the contributing element is positive and sufficient.
- *BREAK contribution*: The contribution of the contributing element is negative and sufficient.
- *HELP contribution*: The contribution of the contributing element is positive but not sufficient.
- *HURT contribution*: The contribution of the contributing element is negative but not sufficient.
- *SOME+ contribution*: The contribution is positive, but the extent of the contribution is unknown.
- *SOME- contribution*: The contribution is negative, but the extent of the contribution is unknown.
- *EQUAL contribution*: There is equal contribution in both directions.

- *UNKNOWN contribution*: There is some contribution, but the extent and the sense (positive or negative) of the contribution is unknown.

A Softgoal Contribution Structure shows the contributions of softgoals or tasks towards a softgoal. See Figure 6 as an example.

An argumentation structure could attach a belief to any link or node of a model, which denotes the contribution of a belief node to the link or node it attached to, and give some argument for future review and justification. See Figure 7 as an example.

#### 4.4.4.1 Textual notation

<Contribution> ::= **CONTRIBUTION** [<Contribution Identifier> **IS**]

<Contributors> **HAS** <Contribution type> **CONTRIBUTION-TO** <Contributee>

<Contributors>::= <Contributor > {"," <Contributor>}$_0^n$

<Contributee > ::= <Softgoal Reference> | <Link Reference> |<Belief Reference>

<Contributor > ::= <Softgoal Reference> | <Task Reference>

| <Link Reference> | <Belief Reference>

<Softgoal Reference> ::= **SOFTGOAL** <Softgoal Name>

<Belief Reference>::= **BELIEF** <Belief Name>

<Link Reference> ::= <Decomposition link Reference> | <Means-ends link Reference>

| <Contribution link Reference> | <Correlation link Reference>

| <Dependency link Reference>

<Decomposition link Reference> ::= **DECOMPOSITION-LINK**< Decomposition Identifier >

<Means-ends link Reference> ::= **MEANS-ENDS-LINK** <Means-Ends Identifier>

<Contribution link Reference> ::= **CONTRIBUTION-LINK** <Contribution Identifier>

<Correlation link Reference> ::= **CORRELATION-LINK** <Correlation Identifier>

<Dependency link Reference> ::= **DEPENDENCY-LINK** <Dependency Identifier>

< Contribution Type> ::= Break | Hurt | Some- | Unknown | Equal

| Some+ | Help | Make | And | Or

#### 4.4.4.2 Graphical notation

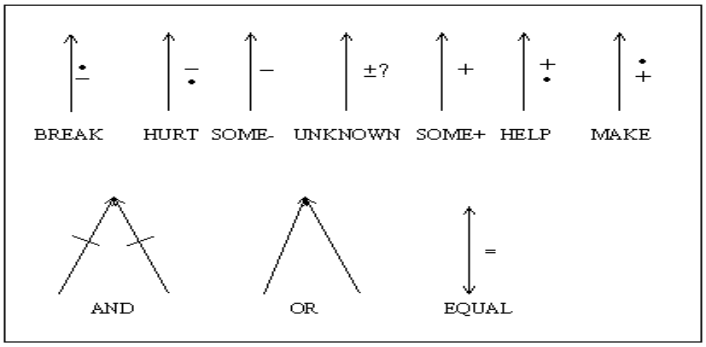
< Contribution> : :=

<Contributee Element>

{**IS CONNECTED TO** <Contributor Element>

**BY** < Contribution Link >}$_0^n$

<Contribution Link> : :=



#### 4.4.4.3 XML definition

```
<!ELEMENT contribution (contributee, contributor, contribution-type)>

<!ATTLIST contribution
            contribution-id    ID                  #REQUIRED>

<!ELEMENT contributee (softgoal-ref | belief-ref | link-ref)>

<!ELEMENT contributor (task-ref | softgoal-ref| belief-ref | link-ref)>

<!ELEMENT contribution-type EMPTY>

<!ATTLIST contribution-type
                    contri-type    (break |
                                    hurt |
                                    some-negative |
                                    unknown |
                                    equal |
                                    some-positive |
                                    help |
                                    make |
                                    and |
                                    or)             #REQUIRED>

<!ELEMENT goal-ref EMPTY>
<!ATTLIST goal-ref
            goal-id-ref              IDREF        #REQUIRED>

<!ELEMENT softgoal-ref EMPTY>
<!ATTLIST softgoal-ref
            softgoal-id-ref          IDREF        #REQUIRED>

<!ELEMENT task-ref EMPTY>
<!ATTLIST task-ref
            task-id-ref              IDREF        #REQUIRED>

<!ELEMENT resource-ref EMPTY>
<!ATTLIST resource-ref
            resource-id-ref          IDREF        #REQUIRED>

<!ELEMENT belief-ref EMPTY>
<!ATTLIST belief-ref
            belief-id-ref            IDREF        #REQUIRED>

<!ELEMENT link-ref (means-ends-ref | decomposition-ref
                  | contribution-ref | correlation-ref
                  | dependency-ref)>

<!ELEMENT means-ends-ref EMPTY>
<!ATTLIST means-ends-ref
            means-ends-id-ref        IDREF        #REQUIRED>

<!ELEMENT decomposition-ref EMPTY>
<!ATTLIST decomposition-ref
            decomposition-id-ref     IDREF        #REQUIRED>

<!ELEMENT contribution-ref EMPTY>
<!ATTLIST contribution-ref
            contribution-link-id-ref IDREF        #REQUIRED>

<!ELEMENT correlation-ref EMPTY>
<!ATTLIST correlation-ref
            correlation-link-id-ref  IDREF        #REQUIRED>

<!ELEMENT dependency-ref EMPTY>
<!ATTLIST dependency-ref
            dependency-id-ref        IDREF        #REQUIRED>
```

#### 4.4.4.4 Example

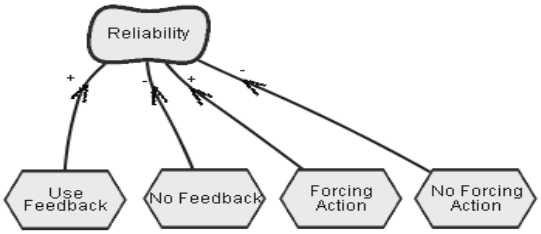



**Figure 6.** Softgoal contribution structure

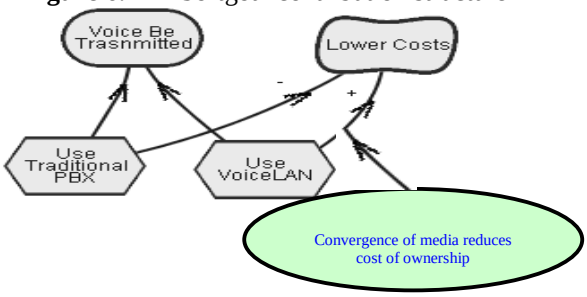



**Figure 7.** Argumentation structure

### 4.5 Actor

Actor statement defines an actor and its attributes.

**ACTOR** <Actor Name> <Actor Description> [<Attributes>]

An actor is an active entity that carries out actions to achieve goals by exercising its know-how. Graphically, an actor may optionally have a boundary, with intentional elements inside.

#### 4.5.1.1 Textual notation

<Actors>::= <Actor > { <Actor>}$_0^n$

<Actor>::= **ACTOR** <Actor Name> [<Informal Textual Description>] [<Attributes>]

#### 4.5.1.2 Graphical notation

<Actor> : :=

<Actor Symbol>

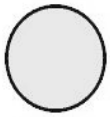
**CONTAINS** <Actor Name> [<Attributes>]

[**IS ASSOCIATED WITH**

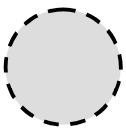
<Actor Boundary Symbol>

**CONTAINS**<Intentional Elements > <Intentional Links> ]

<Actor Symbol> ::=



<Actor Boundary Symbol>::=



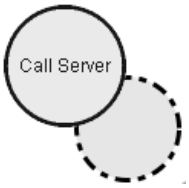
#### 4.5.1.3 XML definition

```
<!ELEMENT actors (actor)+ >

<!ELEMENT actor (attributes?)>

<!ATTLIST actor
              actor-id              ID              #REQUIRED
              actor-name            CDATA           #IMPLIED
              description           CDATA           #IMPLIED>
```

#### 4.5.1.4 Example

Actors are shown as a circle with the name of the actor inside in graphical representations. The boundary of an actor is shown as a grey shadow with the actor icon inside or nearby. For example, within a VoiceLAN environment, an actor "Call Server" on the LAN provides the call control functionality normally provided by a PBX. Figure 8 is an example of actor structure.

In the component base, each intentional element has an attribute named as "**Owner"** by default:




… …

**owner**: <Actor Name>

… …

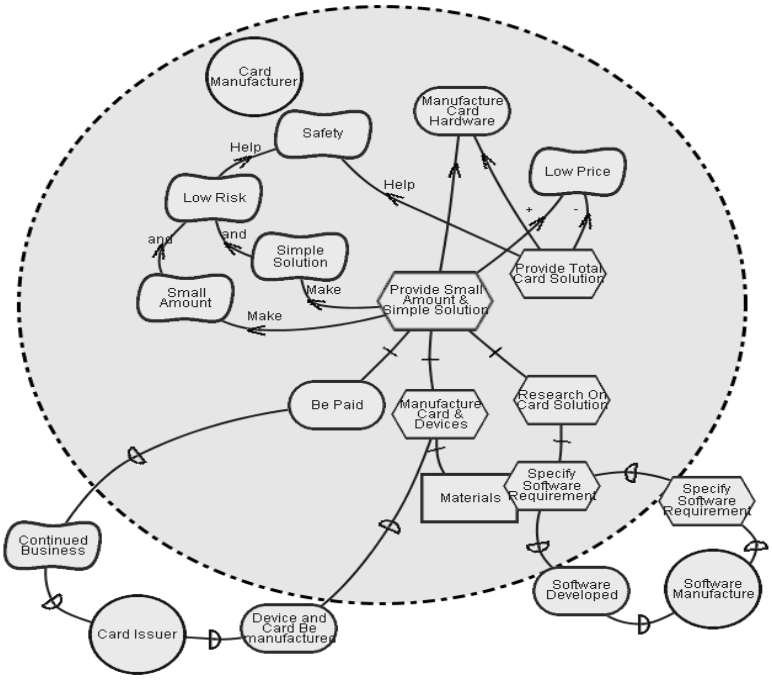



**Figure 8.** Actor with intentional elements and links

## 4.6 Dependency

The Dependency statement of GRL describes an intentional relationship between two actors, i.e., one actor (<Depender>) depends on another actor (<Dependee>) on something (<Dependum>).

**DEPENDENCY** [<Dependency Identifier>] <Depender> **DEPENDES-ON** <Dependee> **FOR** <Dependum>

A dependency link connects (an intentional element of) the depender actor with (the intentional element of) another actor it depends on.

#### 4.6.1.1 Textual notation

<Dependency> ::= **DEPENDENCY** [<Dependency Identifier> **IS**]

<Depender> **DEPENDES-ON** <Dependee> **FOR** <Dependum>

<Depender> : := **DEPENDER** <Actor Name> [“.”<Sub-Element>]

<Dependee> : := **DEPENDEE** <Actor Name> [“.” <Sub-Element >]

<Dependum> : := **DEPENDUM** <Sub-Element>

#### 4.6.1.2 Graphical notation

<Dependency Link> : :=

<Depender> **IS CONNECTED TO** <Dependum>

**IS CONNECTED TO** <Dependee>

**BY** <Dependency Link>

<Depender> : := <Actor > | <Intentional element>

<Dependee> : := <Actor > | <Intentional element>

<Dependum> : := <Intentional element>

<Dependency Link> : :=

#### 4.6.1.3 XML definition

```
<!ELEMENT dependency (depender, dependum, dependee)>
<!ATTLIST dependency
               dependency-id        ID              #REQUIRED>

<!ELEMENT depender
           (goal-ref | softgoal-ref | task-ref | resource-ref)?>
<!ATTLIST depender
               actor-id-ref        IDREF              #REQUIRED>

<!ELEMENT dependum
           (goal-ref | softgoal-ref | task-ref | resource-ref)>

<!ELEMENT dependee
           (goal-ref | softgoal-ref | task-ref | resource-ref)?>
<!ATTLIST dependee
               actor-id-ref        IDREF              #REQUIRED>
```

#### 4.6.1.4 Example

In Figure 9, a model composed of many actors, and with dependencies is shown. With this kind of model, the intentional relationships between actors can be better captured and analyzed. There are altogether four types of dependencies, which permit different degree of freedom for the depended actor to make decision.

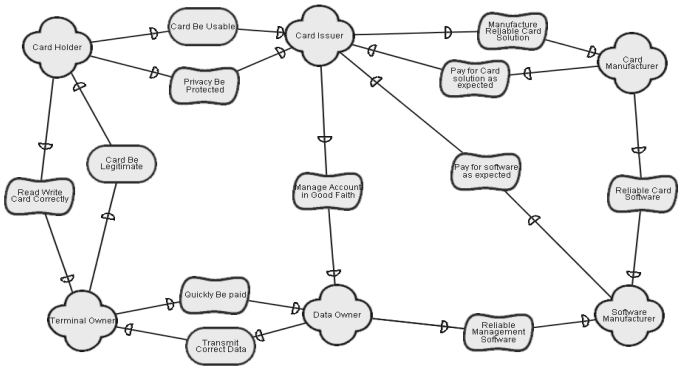



**Figure 9.** A model composed of many actors and their dependencies

## 4.7 Correlations

Correlations allow for expressing knowledge about interactions between intentional elements in different categories, and to encode such knowledge. A correlation link is the same as a contribution link except that the contribution is not an explicit desire, but is a side effect. This type of relationship is captured by a CORRELATION statement, which is defined as follows:

**CORRELATION** <Correlation Identifier> **IS**

<Correlator > **HAS** <Correlation Type> **CONTRIBUTION-TO** <Correlatee>

The effect of all incoming correlation links on a softgoal may need to be evaluated by the user on a case-by-case basis.

#### 4.7.1.1 Textual notation

<Correlation> ::= **CORRELATION** [<Correlation Identifier> **IS**]

<Correlator > **HAS** <Correlation Type> **CONTRIBUTION-TO** <Correlatee>

<Correlatee> ::= <Softgoal Reference>

<Correlator> ::= <Softgoal Reference> | <Task Reference>

<Correlation Type> ::= Break | Hurt | Some- | Some+ | Help | Make | Equal

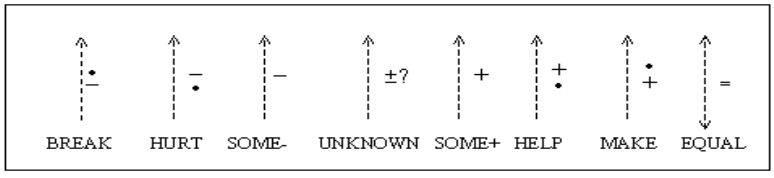
#### 4.7.1.2 Graphical notation

<Correlation> : :=

<Correlatee> **IS CONNECTED TO** <Correlator>

**BY** <Correlation Link>

<Correlation Link> : :=



#### 4.7.1.3 XML definition

```
<!ELEMENT correlation (correlatee, correlator, correlation-type)>

<!ATTLIST correlation
            correlation-id    ID                  #REQUIRED>

<!ELEMENT correlatee (softgoal-ref)>

<!ELEMENT correlator (task-ref | softgoal-ref)>

<!ELEMENT correlation-type EMPTY>

<!ATTLIST correlation-type
               correlation-type  (break |
                                  hurt |
                                  some-negative |
                                  unknown |
                                  some-positive |
                                  help |
                                  make |
                                  equal)          #REQUIRED>
```

#### 4.7.1.4 Example

In Figure 10, a model of basic model structure is given, which is made up of one softgoal structure, it can also made up of several model structures. Each of these structures is a category of requirement. They interact through <Correlation Structure Series>. See Figure 11 as an example.

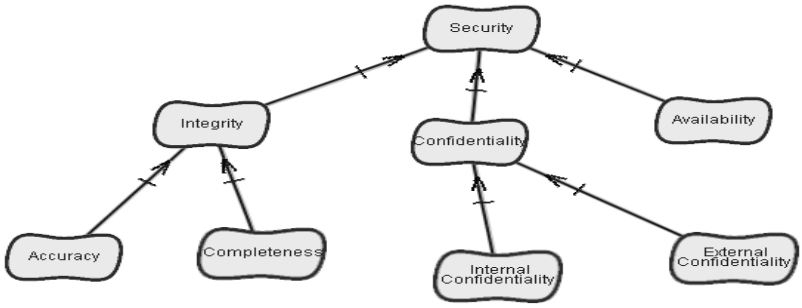



**Figure 10.** A Model composed of basic model structure

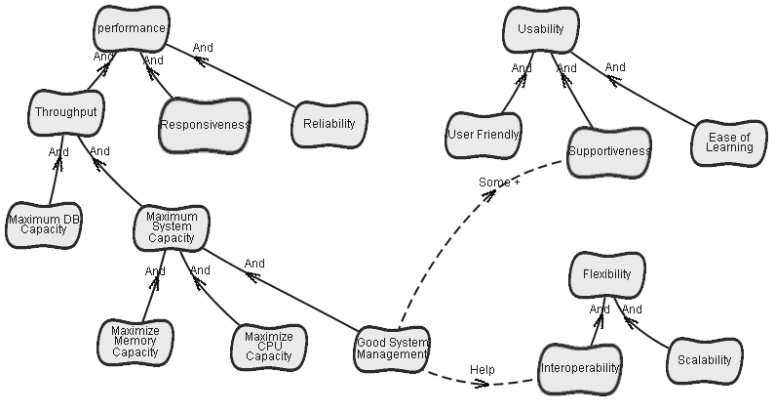



**Figure 11.** Model structure series connected with correlations

### 4.8 Requirement knowledge base

Besides expressing requirement model of an enterprise, GRL are also used to express general or domain-specific knowledge on NFRs and architectural design. General knowledge includes the NFRs and architecture design rules that fit into any application domain, such as “Encryption contributes to Security positively”, “Responsiveness is necessary for a good system performance”, etc. In contrast, domain-specific knowledge only fits into certain application domain. For

example, the performance of a telecommunication system means the speed and capacity during peak data traffic periods, but the performance of an information system for library management is its responsiveness and availability. Establishing a knowledge base in various conceptual layers makes it possible to reuse existing requirement models and engineering experiences.

Requirement models are composed of elements and some basic building structures, accordingly, the reusable knowledge in KB can be classified into two categories: concept base and rule base. The concept base stores the reusable concepts, and their relationships; while the rule base stores the reusable model construction rules. In the concept base, as the accumulated knowledge increases, the generalization and specialization of concepts is unavoidable. Thus, two mechanisms: class inheritance mechanism, and attribute mechanism serve this aim.

The representation of concept and rules, the approaches to manage and use goal-oriented requirement knowledge are to be completed in the future.

# 5 URN-FR language specification

The User Requirements Notation - Functional Requirements (URN-FR) language is based on the Use Case Maps notation (UCMs). URN-FR specifications employ scenario paths to illustrate causal relationships among responsibilities. Furthermore, URN-FR provides an integrated view of behavior and structure by allowing the superimposition of scenario paths on a structure of abstract components. The combination of behavior and structure enables architectural reasoning after which URN-FR specifications may be refined into more detailed models such as MSCs and UML interaction diagrams. These detailed models may be transformed into state machines in SDL or UML-RT statechart diagrams and finally into concrete implementations (possibly through automated code generation). Validation, verification, performance analysis, interaction detection, and test generation could be performed at all stages. Thus, the URN-FR language enables a seamless transition from the informal to the formal by bridging the modeling gap between requirements (use cases) and design in an explicit and visual way. URN-FR delays 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 URN-FR is to provide the right degree of formality at the right time in the development process.

URN-FR specifications identify input sources and output sinks as well as describe the required inputs and outputs of a scenario but 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 URN-FR language 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.

URN-FR is a specification language intended for specifiers 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 programmers 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 URN-FR. Others such as prioritized and annotated are easily incorporated.

URN-FR treat scenario paths as first class model entities and therefore build the foundation to more formally facilitate reusability of scenarios and behavioral patterns across a wide range of architectures.

## 5.1 URN-FR specifications

A URN-FR specification is composed of a collection of (top-level) `root-maps` and, possibly, of a collection of `plug-in-maps`, with their bindings. Both root maps and plug-in maps define functional requirements models (`fr-model`). Root maps however describe the top-level functional requirements whereas plug-in maps describe (reusable or shared) partial requirements or sub-requirements. An `fr-model` contains sub-specifications for its scenario paths and for its component structure. It also has several attributes: an identifier, a name, a title, and a description.

#### 5.1.1.1 XML definition

```
<!ELEMENT urn-fr-spec (root-maps, plug-in-maps?, plug-in-bindings?)>

<!ELEMENT root-maps (fr-model)+>

<!ELEMENT plug-in-maps (fr-model)*>

<!ELEMENT fr-model (path-spec?, structure-spec?)>

<!ATTLIST fr-model
                fr-model-id    ID      #REQUIRED
                fr-model-name  CDATA   #REQUIRED
                title          CDATA   "No title"
                description    CDATA   #IMPLIED >
```

#### 5.1.1.2 Graphical notation

No graphical notation is required for these DTD elements.

## 5.2 Path notation

### 5.2.1 Hypergraph

This section defines the URN-FR path notation used for the definition of causal scenarios (also called *maps*) included in the element `path-spec`. A path specification of an `fr-model` details the underlying *hypergraph* that represents the causal scenarios, i.e. the paths. A hypergraph is a graph structure specifying all the elements (called *hyperedges*) that make up the paths and their interconnections. It also includes the specification of path branching forks as well as constraints on the plug-in maps (i.e. the sub-maps) that can be connected. Hyperedges contain the basic path constructs: start and end points; waiting places; responsibility references; OR-joins and OR-forks; AND-joins and AND-forks (synchronizations); loops; aborts; connections; stubs; performance and goal annotations; and empty segments. All hyperedges have an identifier, a name, a description, and the XY coordinates of the hyperedge on the map workspace (*fx* and *fy* are real numbers between 0 and 1). Attributes *lx* and *ly* specify the location of the hyperedge name (if any) relative to the location of the hyperedge (i.e. $-fx \leq lx \leq 1-fx$, and $-fy \leq ly \leq 1-fy$).

A hypergraph has a list of connections between hyperedges (`hyperedge-connection`). Each such connection is a *source-hyperedge* attribute and a list of references to target hyperedges (`hyperedge-ref`). These references (of attribute type **IDREF**) must refer to existing hyperedge identifiers (of attribute type **ID**).

#### 5.2.1.1 XML definition

```
<!ELEMENT path-spec (hypergraph)?>

<!ELEMENT hypergraph (hyperedge*, hyperedge-connection*,
                      path-branching-spec*, enforce-bindings*)>

<!ELEMENT hyperedge ( (start | end-point | responsibility-ref |
            fork | join | synchronization | loop | stub |
            waiting-place | abort | connect | timestamp-point |
            goal-tag | empty-segment)) >

<!ATTLIST hyperedge
                hyperedge-id      ID       #REQUIRED
                hyperedge-name    CDATA    #IMPLIED
                fx                NMTOKEN  #IMPLIED
                fy                NMTOKEN  #IMPLIED
                lx                NMTOKEN  #IMPLIED
                ly                NMTOKEN  #IMPLIED
                description       CDATA    #IMPLIED >

<!ELEMENT hyperedge-connection (hyperedge-ref)*>

<!ATTLIST hyperedge-connection
                source-hyperedge IDREF    #REQUIRED >

<!ELEMENT hyperedge-ref EMPTY>

<!ATTLIST hyperedge-ref
                hyperedge-id      IDREF    #REQUIRED >
```

#### 5.2.1.2 Graphical notation

A line (spline) is used to represent connected path elements (hyperedges) that belong to a causal scenario. The hyperedges themselves have different visual representations depending on their nature, and they will be defined formally in the next sub-sections.

#### 5.2.1.3 Example

The next two figures are used as an introduction example to the path notation. The core elements are present in the simple map shown in Figure 12. This causal scenario path represents a simplified call connection initiated through the start point req. The system first checks whether the call should be allowed (responsibility chk) and then verifies whether the called party is busy or idle (vrfy). The assumption in this scenario (i.e. a precondition) is that the called party is idle. Then, the system status is updated (upd) and a resulting ringing event occurs at the end point (ring). The causal path is shown as a line (usually a spline) that connects the various hyperedges.

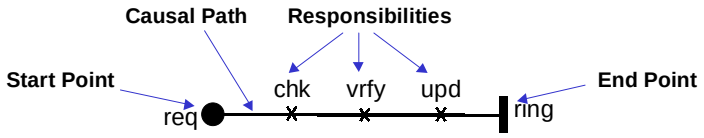
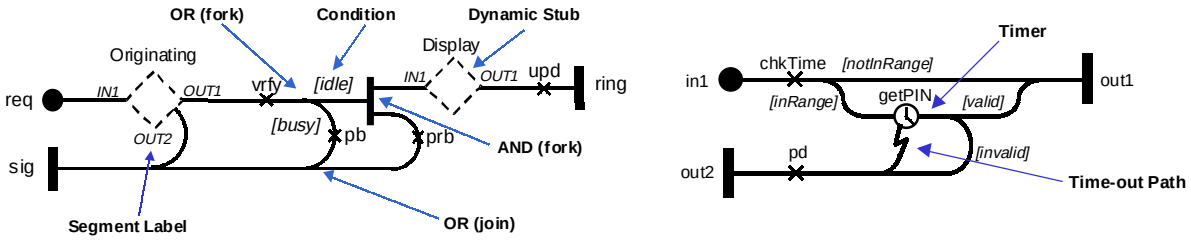



**Figure 12.** Basic URN-FR notation elements

URN-FR can also help structuring and integrating scenarios in various ways, e.g. sequentially, as alternatives (with OR-forks/joins) or concurrently (with AND-forks/joins). Conditions can be attached to alternative paths. Stubs are containers for sub-maps called plug-ins and can also be used to integrate, structure, and reuse scenarios. Figure 13(a) extends the simple connection example by using some of these notation elements. The verification responsibility (vrfy) may cause the selection of the idle path, which then splits into two concurrent paths for ringing (ring) and for signaling (sig) the occurrence of a prepared ringback reply (prb). Else, the busy path could be selected, which would result in the signaling of a prepared busy reply (pb). Figure 13(b) presents a potential plug-in map for the Originating stub from Figure 13(a). This *TEENLINE* plug-in checks the current time (chkTime) and, if in the predefined range, requires a valid personal identification number (PIN) to be provided in a timely fashion for the call initiation to continue. If an invalid PIN is provided, or if a time-out occurs, then a denied reply is prepared (pd). The input/output path segments attached to stubs are labeled for binding plug-in start/end points. The binding relationship is {<IN1, in1>, <OUT1, out1>, <OUT2, out2>} connects the stub path segments of the parent map to the start/end points of the plug-in.




a) *Call initiation* root map

b) *TEENLINE* plug-in map

**Figure 13.** Extension of the call connection example and additional URN-FR notation elements

All these elements are defined individually in the remaining sub-sections.

### 5.2.2 Start points

A start point is where scenarios are caused, upon the arrival of associated triggering events and/or the satisfaction of associated preconditions. A `start` hyperedge should be connected to only one target hyperedge. It has no source hyperedge except when linked to an `end-point` by a `connect` hyperedge. A `triggering-event-list` gives the set of events that can initiate the causal scenario along a path. The required events can be composed in many ways (user-defined), but by default one of them is sufficient for triggering the scenario. The `precondition-list` must also be satisfied in order for the scenario to be enabled. Several attributes used for performance analysis can be attached to a start point:

- *`mean`*: parameter for exponential and Erlang arrival distributions.
- *`value`*: parameter for deterministic arrival distribution.
- *`low`*, *`high`*: parameters for uniform arrival distribution.
- *`kernel`*: parameter for Erlang arrival distribution.
- *`expert-distribution`*: parameter for expert arrival distribution (user-defined).

#### 5.2.2.1 XML definition

```
<!ELEMENT start (triggering-event-list?, precondition-list?)?>

<!ATTLIST start
                arrival     (exponential | deterministic
                             | uniform | erlang
                             | expert | none)          "none"
                mean                NMTOKEN            #IMPLIED
                value               NMTOKEN            #IMPLIED
                low                 NMTOKEN            #IMPLIED
                high                NMTOKEN            #IMPLIED
                kernel              NMTOKEN            #IMPLIED
                expert-distribution CDATA              #IMPLIED >

<!ELEMENT triggering-event-list (event)*>

<!ATTLIST triggering-event-list
                composition         CDATA              "OR" >
```


#### 5.2.2.2 Graphical notation

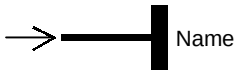
A start point is shown as a filled circle at the beginning of a path, with its *`hyperedge-name`* attribute as a label.

### 5.2.3 End points

Scenario effects are represented by end points, which describe resulting events and/or post-conditions. An `end-point` hyperedge should be connected to only one source hyperedge. It has no target hyperedge except when linked to a `start` or to a `waiting-place` by a `connect` hyperedge. A `resulting-event-list` gives the set of events that result from the completion of the causal scenario path. These events can be composed or selected in many ways (user-defined), but by default all of them are output. The `post-condition-list` is satisfied once the sequence is completed.

#### 5.2.3.1 XML definition

```
<!ELEMENT end-point (resulting-event-list?, postcondition-list?)?>

<!ELEMENT resulting-event-list (event)*  >

<!ATTLIST resulting-event-list
                composition         CDATA               "AND" >
```




#### 5.2.3.2 Graphical notation

An end point is shown as a thick bar with its *hyperedge-name* attribute as a label.

### 5.2.4 Events and conditions

Conditions have a name and a user-defined description. No specific syntax or data language is specified at the moment, although in the future conditions should be described in a language compatible with the *data-language* attribute of the urn-spec element. Conditions can be grouped as lists of preconditions and post-conditions. These conditions can be composed in a user-defined way, but by default they must all be satisfied ("AND" composition). An event is an element with a name and a description.

#### 5.2.4.1 XML definition

```
<!ELEMENT postcondition-list (condition)*>

<!ATTLIST postcondition-list
                composition CDATA   "AND" >

<!ELEMENT precondition-list (condition)*>

<!ATTLIST precondition-list
                composition CDATA   "AND" >

<!ELEMENT condition EMPTY>

<!ATTLIST condition
                name        NMTOKEN  #REQUIRED
                description CDATA    #IMPLIED >

<!ELEMENT event EMPTY>

<!ATTLIST event
                name        NMTOKEN  #REQUIRED
                description CDATA    #IMPLIED >
```

#### 5.2.4.2 Graphical notation

There is no Graphical notation defined for these elements.

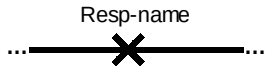
### 5.2.5 Responsibility references

A responsibility reference refers, through its *resp-id* attribute, to a responsibility defined in responsibility-definitions. An optional *arrow-position* attribute indicates the position of the arrow when the responsibility is dynamic (see dynamic-resp in Section 5.5). The responsibility-ref hyperedge is connected to only one source hyperedge and to one target hyperedge.

#### 5.2.5.1 XML definition

```
<!ELEMENT responsibility-ref EMPTY>

<!ATTLIST responsibility-ref
                resp-id        IDREF     #REQUIRED
                arrow-position NMTOKEN   #IMPLIED >
```

#### 5.2.5.2 Graphical notation



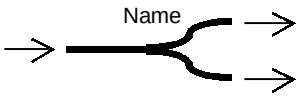

A responsibility reference is shown as a cross on a path with the *resp-name* attribute of the referenced responsibility as a label. Responsibilities that are dynamic are shown with arrows, as described in Section 5.5.

### 5.2.6 OR-forks and OR-joins

OR-forks represent on a path where scenarios split as two or more alternative paths. A `fork` has several target hyperedges and only one source hyperedge. OR-joins capture the merging of two or more independent scenario paths. A `join` has several source hyperedges and only one target hyperedge. Both elements have an *orientation* attribute which describes, in degrees (0º for right, 90º for up, etc.), the orientation of the construct. Conditions can be attached to forked paths, as specified in `path-branching-spec` (Section 5.2.15).

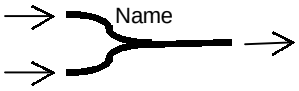
#### 5.2.6.1 XML definition

```
<!ELEMENT fork EMPTY>

<!ATTLIST fork
                orientation NMTOKEN #IMPLIED >

<!ELEMENT join EMPTY>

<!ATTLIST join
                orientation NMTOKEN #IMPLIED >
```




#### 5.2.6.2 Graphical notation

An OR-fork is shown as splitting paths (two or more). If specified, the *hyperedge-name* attribute of this hyperedge is shown as a label.




An OR-join is shown as merging paths (two or more). If specified, the *hyperedge-name* attribute of this hyperedge is shown as a label.

### 5.2.7 AND-forks, AND-joins, and synchronizations

AND-forks represent on a path where scenarios split as two or more concurrent paths (i.e. the concurrency level is increased). AND-joins capture the synchronization of two or more concurrent scenario paths (i.e. the concurrency level is decreased). Both AND-joins and AND-forks are special cases of the `synchronization` element, which has several source and target hyperedges. An explicit cardinality (1:N, N:1, 2:3, etc.) can also be associated to a synchronization to specify how many instances of the source path need to synchronize and/or how many instances of the target path are created concurrently. This element has an *orientation* attribute which describes its orientation, in degrees (0º for right, 90º for up, etc.).

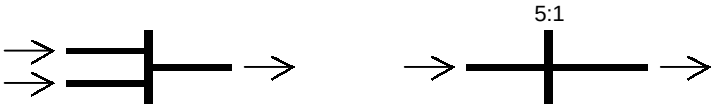
#### 5.2.7.1 XML definition

```
<!ELEMENT synchronization EMPTY>

<!ATTLIST synchronization
                cardinality-source  NMTOKEN #IMPLIED
                cardinality-target  NMTOKEN #IMPLIED
                orientation         NMTOKEN #IMPLIED >
```

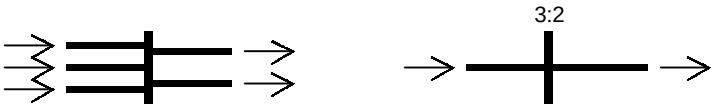
#### 5.2.7.2 Graphical notation

A synchronization is shown as a thin bar, generally perpendicular to the path(s). If specified, the cardinality is shown in the format cardinality-source:cardinality-target. If specified, the *hyperedge-name* attribute of this hyperedge can also be shown as a label.


The figure above presents examples of AND-forks that have one source path and multiple target paths.




The figure above presents examples of AND-joins that have multiple source paths and one target path.




The figure above presents examples of general synchronizations with multiple source paths and multiple target paths.

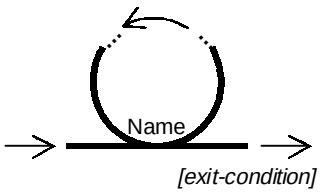
### 5.2.8 Loops

Loops are captured explicitly by the `loop` element. It has two source hyperedges, which represent the original path and the end of the looping path. It also has two target hyperedges, which represent the continuation of the original path and the beginning of the looping path. An *`exit-condition`* attribute specifies the condition under which the loop is exit (the looping path may even never been entered). The looping path may contain any other hyperedge, including other loops. This element has an *`orientation`* attribute which describes its orientation, in degrees (0º for right, 90º for up, etc.).

#### 5.2.8.1 XML definition

```
<!ELEMENT loop EMPTY>

<!ATTLIST loop
                orientation    NMTOKEN #REQUIRED
                exit-condition CDATA   #IMPLIED >
```

#### 5.2.8.2 Graphical notation




The loop is shown as a path element with two incoming path segments followed immediately by two outgoing path segments (similar to an OR-join followed by an OR-fork). If specified, the *`hyperedge-name`* attribute of this hyperedge is shown as a label. The exit condition can also be shown between square brackets near the exit path.

### 5.2.9 Stubs

Stubs are containers for plug-in maps (i.e. sub-maps). Stubs can be of *`type`* static or of *`type`* dynamic. While static stubs contain only one plug-in, dynamic stubs may contain multiple plug-ins whose selection can be determined at run-time according to a *`selection-policy`*. Such a policy can make use of preconditions, assertions, run-time information, composition operators, etc. in order to select the plug-in(s) to use. Selection policies are described with a (formal or informal) language suitable for the context where they are used. The plug-in maps are sub-maps that describe locally how a feature modifies the parent causal paths. Multiple levels of stubs and plug-ins can be used. Stubs that involve negotiations with components other than the one to which the stub is allocated are called *`shared`* stubs.

A stub can have several entry points and exit points, connected respectively to source and target hyperedges. The paths segments that are connected to the stub need to be bound to the paths of the plug-ins in order to express continuity. This is done through explicit binding in `plug-in-bindings` (Section 5.4.1). Bindings can be enforced to ensure particular path continuity, even in the presence of a plug-in (see `enforce-bindings`, Section 5.4.2). `stub-entry-list` contains identifiers for stub entry points connected to source hyperedges, whereas `stub-exit-list` contains identifiers for stub exit points connected to target hyperedges.

Stubs may also contain preconditions, post-conditions, and a specification of service requests for performance requirements (Section 5.6.1).

#### 5.2.9.1 XML definition

```
<!ELEMENT stub (stub-entry-list, stub-exit-list, precondition-list?,
                postcondition-list?, service-request-spec?)>

<!ATTLIST stub
                type              (static | dynamic)  "static"
                shared            (yes | no)          "no"
                selection-policy  CDATA               #IMPLIED>

<!ELEMENT stub-entry-list (stub-entry)+>

<!ELEMENT stub-entry EMPTY>

<!ATTLIST stub-entry
                stub-entry-id   ID      #REQUIRED
                hyperedge-id    IDREF   #REQUIRED >

<!ELEMENT stub-exit-list (stub-exit)+>

<!ELEMENT stub-exit EMPTY>

<!ATTLIST stub-exit
                stub-exit-id    ID      #REQUIRED
                hyperedge-id    IDREF   #REQUIRED >
```

#### 5.2.9.2 Graphical notation


Stubs are shown as diamonds connecting input path segments to output path segments. The *hyperedge-name* attribute of this hyperedge is shown as a label. Diamonds for dynamic stubs are drawn with dotted lines.


Shared stubs are shown with two overlapping diamonds, and they can be either static or dynamic.

### 5.2.10 Waiting places and timers

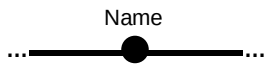
A waiting place is an element on a path where the causal flow stops until it is restarted by a specific event coming from the environment or from some other path via a `connect` element. The `triggering-event-list` gives the set of events that can restart the causal flow on a path. The `precondition-list` must be satisfied in order for the flow to restart. A *`timer`* is a special waiting place where the flow resumes along the continuation path if the triggering event arrives in a timely fashion, otherwise the flow continues on a timeout path. The *`wait-type`* attribute is a user-defined description of the type of waiting (e.g. it could collect the triggering events before the causal flow gets to the waiting place).

A `waiting-place` has usually one source hyperedge and one target hyperedge. However, it can have two target hyperedges when there is a timeout path (the latter uses an `abort` hyperedge, see Section 5.2.11), and more than one source hyperedges when linked to `connect` hyperedges (Section 5.2.12).

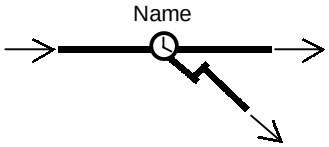
#### 5.2.10.1 XML definition

```
<!ELEMENT waiting-place (triggering-event-list?, precondition-list?)?>

<!ATTLIST waiting-place
                timer           (yes | no)    "no"
                wait-type       CDATA       #IMPLIED >
```

#### 5.2.10.2 Graphical notation




Due to their similarity with start points, waiting places are also shown as filled circles, but this time located *on* a path. The *`hyperedge-name`* attribute of this hyperedge is shown as a label.




Timers are shown with a *clock* symbol. The timeout path that exits from the timer is shown with a broken line, to distinguish it from the initial continuation path.

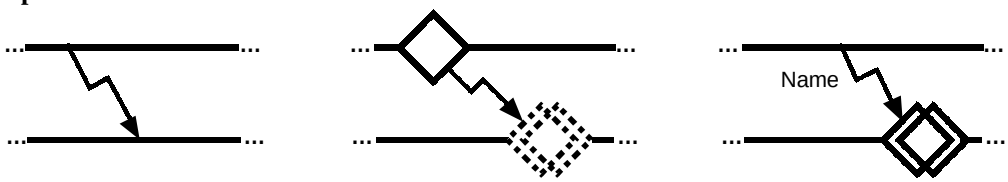
### 5.2.11 Aborts

The abort describes, in an exception-like manner, that one scenario path aborts or disables the flow along another path. It is also used to describe the timeout path exiting from a timer. The abort has one source hyperedge and one target hyperedge, which can be empty segments or stubs. In the case of a timeout path, the source hyperedge is a waiting place (timer) and the target hyperedge is an empty segment.

#### 5.2.11.1 XML definition

```
<!ELEMENT abort EMPTY>
```

#### 5.2.11.2 Graphical notation




The abort is shown as a broken arrow linking an empty point or a stub to an empty point or a stub. The head of the arrow points towards the target hyperedge. When defined, the *hyperedge-name* attribute of this hyperedge is shown as a label.

When used to represent a timeout path, the abort is shown as a broken line instead of as an arrow (Section 5.2.10).

### 5.2.12 Connections

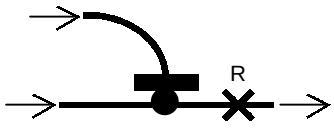
Connections describe explicit interactions between two different scenario paths. Connections capture synchronous interactions by linking an end point to a start point or to a waiting place (or timer). Such interactions trigger the target start point or waiting place (or timer). This triggering can also be achieved in-passing (without waiting) through asynchronous interactions. In this case, the source hyperedge is an empty segment. The connection has one source hyperedge (end point or empty segment) and one target hyperedge (start point or waiting place).

#### 5.2.12.1 XML definition

```
<!ELEMENT connect EMPTY>
```

#### 5.2.12.2 Graphical notation

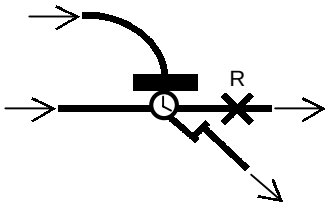
A connect hyperedge does not have a graphical representation as an element. It is however shown as the juxtaposition of an end point with a start point of waiting place (for synchronous interactions) or as the juxtaposition of an empty path segment with a start point of waiting place (for asynchronous interactions).



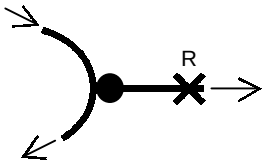

#### 5.2.12.3 Examples

The example above shows a synchronous interaction on a waiting place. Responsibility R cannot be performed until the waiting place is triggered by the end point.


The example above shows a synchronous interaction on a start point. Responsibility R cannot be performed until the start point is triggered by the end point.




The example above shows a synchronous interaction on a timer. Responsibility R cannot be performed until the timer is triggered by the end point, in a timely fashion. If the triggering event does not arrive in time, then the timeout path is followed.




The example above shows an asynchronous interaction on a start point. Responsibility R cannot be performed until the start point is triggered in passing by the empty path segment.

### 5.2.13 Performance and goal annotations

Specific hyperedges can be use to annotate paths for performance requirements (`timestamp-point`) or functional goals (`goal-tag`). These elements have one source hyperedge and one target hyperedge. They are defined with the other elements related to annotations in Section 5.6.

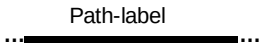
### 5.2.14 Empty segments

An empty segment is used to add *`characteristics`* to a path such as the indication of a failure point (where the causal flow could stop for a non-specified reason, which is useful to describe when robustness needs to be modeled), of the direction of the causal flow, or that the responsibilities immediately preceding and following it are shared. Shared responsibilities explicitly imply the existence of a complex communication mechanism (e.g. multiple exchanges of messages satisfying some constraints, protocols, or negotiation rules) used across components.

An empty segment may also specify preconditions for the next hyperedge or post-conditions of the previous hyperedge. An optional *`path-label`* can be defined for this element. The empty segment has one source hyperedge (more when linked to an abort) and one target hyperedge (more when linked to a connect).

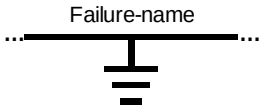
#### 5.2.14.1 XML definition

```
<!ELEMENT empty-segment (precondition-list?, postcondition-list?)?>

<!ATTLIST empty-segment
               path-label      CDATA                       #REQUIRED
               characteristics (failure-point | shared
                                | direction-arrow )        #IMPLIED
               show-label      (yes | no)                  "no" >
```

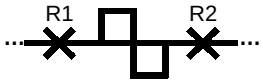



#### 5.2.14.2 Graphical notation

An empty element without any associated attribute does not have any specific graphical representation (the conditions are not shown). This hyperedge however has a position and it can be used to add points along a spline, or to allocate a path segment to a component. In all cases, the *`path-label`* attribute can be displayed, according to the *`show-label`* policy.




An empty point can be characterised as a failure point, which is then shown with a ground symbol.




As shown with the symbol above, an empty point can be used to indicate that two responsibilities (e.g. R1 and R2, usually bounded to two different components) are shared.

In long or ambiguous paths, an empty point can be used to indicate visually the current direction of the causality flow. The direction is shown with an arrow on the path.

#### 5.2.15 Path branching specification

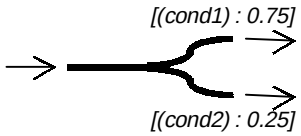
A path branching specification contains the characteristics by which the output branches (referenced by the next *empty-segment*) of an existing OR-fork (referenced by *fork-id*) are selected at run time. The *characteristic* may be described with logical conditions or preconditions. A probability (real number between 0 and 1) can also be assigned to the various branches. The sum of all probabilities for a given OR-fork should be equal to 1. Attributes *lx* and *ly* specify the location of the characteristic relative to the location of the referenced `fork` hyperedge (i.e. $-fx \leq lx \leq 1-fx$, and $-fy \leq ly \leq 1-fy$).

The `path-branching-spec` elements are found after the `hyperedge` elements in a URN-FR description because the former contain references to the latter.

##### 5.2.15.1 XML definition

```
<!ELEMENT path-branching-spec (path-branching-characteristic)*>

<!ATTLIST path-branching-spec
                fork-id    IDREF   #REQUIRED >

<!ELEMENT path-branching-characteristic EMPTY>

<!ATTLIST path-branching-characteristic
                characteristic  CDATA   #IMPLIED
                probability     NMTOKEN #IMPLIED
                empty-segment   IDREF   #REQUIRED
                lx              NMTOKEN #IMPLIED
                ly              NMTOKEN #IMPLIED >
```

##### 5.2.15.2 Graphical notation




Each path branching characteristic is shown in the format [*characteristic* : *probability*]. If the characteristics or probabilities are not both defined, then the format can be simplified to [*characteristic*] or to [:*probability*].

### 5.3 Components and structures

The URN-FR language combines behavioural scenarios with structures of abstract components. This section defines the component notation and its role in a structure.

#### 5.3.1 Component definitions

The `component-definitions` element, found in `definitions`, describes the structural entities of a URN-FR specification, i.e. the components. Components represent, at the requirements level, abstract entities corresponding to actors, processes, objects, containers, agents, and so on. All components have a name, a unique identifier, a description, and a colour (in RGB, 24 bits). Other visual attributes could be added to this element.

Components are divided in two different categories: regular components and pools. Regular components can be of various types, which conform to the *component-notation* attribute found in `urn-spec`. In Buhr's notation, which is used by default in the URN-FR language, the component *type* can be one of the following:

- *Team*: default/generic component, used as a container for sub-components of any type.
- *Process*: active component, which implies the existence a control thread.
- *Object*: passive component, which is usually controlled by a process.
- *Agent*: autonomous component, which act on behalf of other components.

All types of regular components also contain the following boolean attributes:

- *slot*: placeholder for a dynamic component that represents, in a static way, a role that can be populated by actual instances of components at different times.
- *protected*: the execution of causal paths bound to the component is ruled by a mutual exclusion mechanism.
- *replicated*: multiple instances of this component exist simultaneously. The number of instances can be set using a *replication-factor*.

An attribute used for performance analysis may optionally be specified:

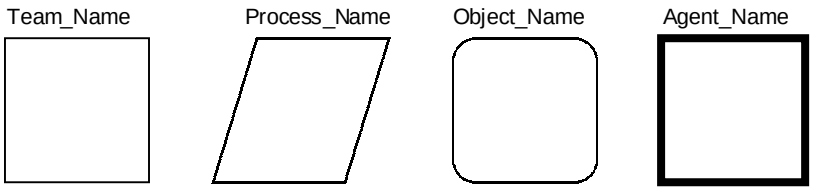
- *processor-id*: reference to a valid processor on which this component executes.

Slots are containers for dynamic components in execution, whereas pools are containers for dynamic components that are not executing. Pools mainly serve the dynamic self-configuring aspect of functional scenarios. A pool can be attributed an arbitrary component type (process, team, object, agent, etc.) or a list of plug-ins usable in dynamic stubs (see the `plug-in-pool` element in Section 5.4.1). Pools can only perform responsibilities that are dynamic, for example, to move components or plug-ins in or out a path (Section 5.5).

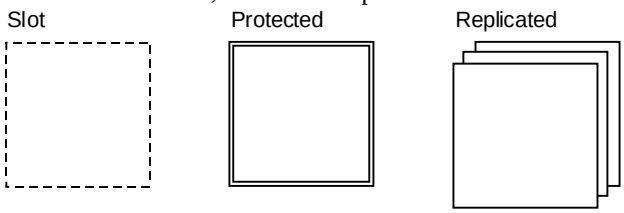
#### 5.3.1.1 XML definition

```
<!ELEMENT component-definitions (component)*>

<!ELEMENT component ((regular | pool))>

<!ATTLIST component
                component-id        ID          #REQUIRED
                component-name      CDATA       #REQUIRED
                description         CDATA       #IMPLIED
                colour              NMTOKEN     #IMPLIED >

<!ELEMENT regular EMPTY>

<!ATTLIST regular
                type                NMTOKEN     "team"
                protected           (yes | no)  "no"
                slot                (yes | no)  "no"
                replicated          (yes | no)  "no"
                replication-factor  NMTOKEN     "1"
                processor-id        IDREF       #IMPLIED >

<!ELEMENT pool EMPTY>

<!ATTLIST pool
                type                NMTOKEN  #REQUIRED >
```

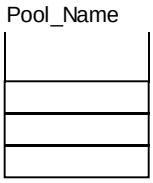
#### 5.3.1.2 Graphical notation




Component definitions do not have a graphical presentation per se. However, references to these components are shown visually (see Section 5.3.2). A team is represented with a rectangle, a process with a parallelogram, an object with a rounded-corner rectangle, and an agent as a rectangle with a thick border. The *component-name* is also displayed on top of the component. If the *colour* attribute is set, then the component border uses this colour.




The boolean attributes also have a graphical representation. Slot components are shown with a dotted border, protected components with a double-line border, and replicated components as a stack with several instances. All these attributes and their graphical can be used simultaneously.




A pool is shown as a partially filled container.

### 5.3.2 Structure specification

The structure specification is part of an `fr-model` and contains references to components, which in turn enumerate the responsibilities and other hyperedges allocated to these components. A `component-ref` element has a unique identifier and a reference to a valid *`component-id`* found in `component-definitions`. It also includes a `responsibility-list`, which references valid responsibility reference hyperedges, as well as an `other-hyperedge-list`, which references valid hyperedges (other than responsibility references) found in the hypergraph. All these referenced hyperedges are said to be *bound* or *allocated* to the component. Component references possess the XY coordinates of the component on the map workspace (*`fx`* and *`fy`* are real numbers between 0 and 1), together with width and height.

Regular components (e.g. teams) may contain sub-components. A sub-component has a *`component-parent`* attribute that references the parent component (a valid *component-ref-id*). Pools are not allowed to contain sub-components.

A component reference is given a user-defined *`role`*, which is a characterization of this particular instance. For example, a component named "User" may be involved in two several roles, such as "Initiator" or "Responder".

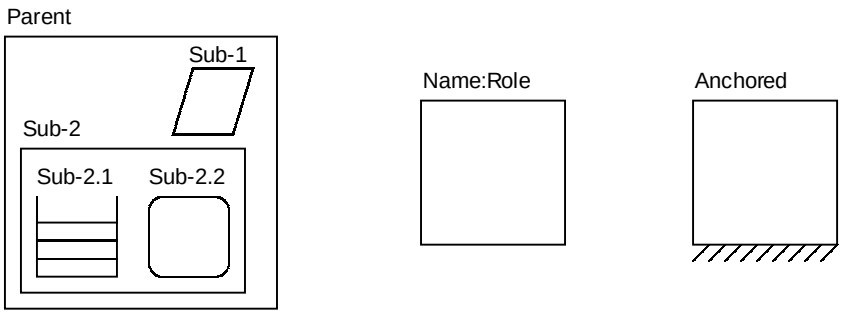
Finally, a component reference can be declared as *`anchored`*. By default, a non-anchored component reference used in a plug-in implies that it is a sub-component of the parent map component where the stub is located. An anchored component reference used in a plug-in is an explicit indication that it is *not* a sub-component of the parent map component where the stub is located (hence, it must be declared somewhere else in the URN-FR specification).

#### 5.3.2.1 XML definition

```
<!ELEMENT structure-spec (component-ref)* >

<!ELEMENT component-ref (responsibility-list?, other-hyperedge-list?)?>

<!ATTLIST component-ref
                component-ref-id      ID          #REQUIRED
                referenced-component  IDREF       #IMPLIED
                component-parent      IDREF       #IMPLIED
                role                  CDATA       #IMPLIED
                fx                    NMTOKEN     #REQUIRED
                fy                    NMTOKEN     #REQUIRED
                width                 NMTOKEN     #REQUIRED
                height                NMTOKEN     #REQUIRED
                anchored              (yes | no)  "no"  >

<!ELEMENT responsibility-list (hyperedge-ref)*>

<!ELEMENT other-hyperedge-list (hyperedge-ref)*>
```

#### 5.3.2.2 Graphical notation

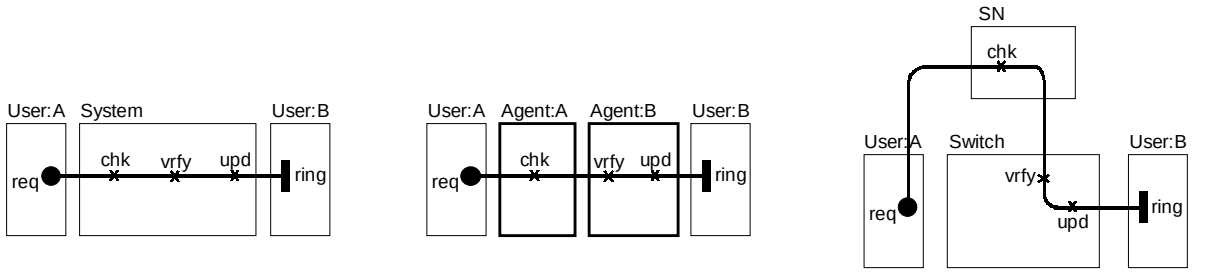
Component references use shapes and patterns corresponding to their respective component definitions (Section 5.3.1). Hyperedge references are allocated to components by superimposing them to component references.




A sub-component is shown as being embedded in its parent component. When non-empty, the role of a component reference is shown in the format `component-name:role`. Anchored component references are shown with a shadow (e.g. using small oblique lines) under the component.

#### 5.3.2.3 Example

Drawing path elements on components is a simple means to evaluate alternative candidate architectures. For instance, the simple call connection scenario (Figure 14) can be bound to various component structures.




a) Scenario path on a simple architecture b) Scenario path on an agent-based architecture c) Scenario path on an IN-based architecture

**Figure 14.** Basic URN-FR notation elements

Figure 14 shows how the same scenario path can be reused on three different example architectures. The first one (a) is a simplistic solution involving the system and two users (roles `A` and `B`). The second architecture (b) splits the responsibilities into two agent instances (roles `A` and `B`). The third solution (c) maps the scenario path to a more conventional architecture based on the Intelligent Network model.

## 5.4 Binding of plug-ins

Plug-in maps are connected by *bindings* to stubs in parent maps. This section presents this binding mechanism together with additional constraints that can be imposed on stub to ensure path continuity (*enforced bindings*).

### 5.4.1 Plug-in bindings

A plug-in map is an `fr-model` that can be substituted to a stub in a parent map. The `plug-in-bindings` element contains a list of plug-in bindings and a list of plug-in pools.

The binding between a plug-in (referred to by *`submap-id`*) and a stub (*`stub-id`*) in its parent map (*`parent-map-id`*) is defined by input and output connection lists. An `in-connection` joins a stub entry identifier (*`stub-entry-id`*, a reference to a valid *`stub-entry-id`* in the stub) with a start hyperedge from the plug-in (valid *`hyperedge-id`*). An `out-connection` joins a stub exit identifier (*`stub-exit-id`*, a reference to a valid *`stub-exit-id`* in the stub) with an end point from the same plug-in (valid *`hyperedge-id`*).

A plug-in can be bound to multiple stubs, which improves the reusability of sub-maps. Start points and end points in a plug-in do not all have to be bound to a stub. Stub entry points and exit points do not all have to be bound either.

A `plug-in-pool` is an identifier container for plug-ins, where plug-ins can be stored or retrieved. The attribute *`fr-model-id`* is a reference to an existing plug-in map identifier whereas *`pool-id`* is a reference to a pool component.

#### 5.4.1.1 XML definition

```
<!ELEMENT plug-in-bindings (plug-in-binding*, plug-in-pool*)? >

<!ELEMENT plug-in-binding (in-connection-list?, out-connection-list?)?>

<!ATTLIST plug-in-binding
              parent-map-id   IDREF   #REQUIRED
              submap-id       IDREF   #REQUIRED
              stub-id         IDREF   #REQUIRED >

<!ELEMENT in-connection-list (in-connection)*>

<!ELEMENT in-connection EMPTY>

<!ATTLIST in-connection
              stub-entry-id   IDREF   #REQUIRED
              hyperedge-id    IDREF   #REQUIRED >

<!ELEMENT out-connection-list (out-connection)*>

<!ELEMENT out-connection EMPTY>

<!ATTLIST out-connection
              stub-exit-id    IDREF   #REQUIRED
              hyperedge-id    IDREF   #REQUIRED >

<!ELEMENT plug-in-pool EMPTY>

<!ATTLIST plug-in-pool
              pool-id         IDREF   #REQUIRED
              fr-model-id     IDREF   #REQUIRED >
```

#### 5.4.1.2 Graphical notation

There is no Graphical notation defined for the binding of plug-ins. Pools have a graphical representation described in Section 5.3.1.

### 5.4.2 Enforced bindings

Required bindings can be enforced while defining a stub in order to preserve path continuity in the parent map. Continuity in a `path-binding` element is expressed as a relation between the entry and exit points of a given stub (i.e. *`stub-entry-id`* and *`stub-exit-id`* must be valid references according to stub *`stub-id`*). Plug-ins whose bindings do not satisfy this constraint will not be selectable.

#### 5.4.2.1 XML definition

```
<!ELEMENT enforce-bindings (path-binding)*>

<!ATTLIST enforce-bindings
              stub-id         IDREF   #REQUIRED >

<!ELEMENT path-binding EMPTY>

<!ATTLIST path-binding
              stub-entry-id   IDREF   #REQUIRED
              stub-exit-id    IDREF   #REQUIRED >
```

#### 5.4.2.2 Graphical notation

There is no Graphical notation defined for these elements.

#### 5.4.2.3 Example

Assume the enforced binding relationship {<IN1, OUT1>} assigned to Originating stub in Figure 13(a). Valid plug-ins are required to provide a path that connects IN1 to OUT1. The plug-in *TEENLINE* in Figure 13(b) possesses such a path according to its binding relationship ({<IN1, in1>, <OUT1, out1>, <OUT2, out2>}), hence it is a valid plug-in.


The *DEFAULT* plug-in shown above may or may not satisfy the same enforced binding relationship. The binding relationship {<IN1, defIn1>, <OUT1, defOut1>} is a valid binding of this plug-in (the hyperedge labels are used here to refer to the elements in the relationships, but the relationships are really using identifiers). However, the binding relationship {<IN1, defIn1>, <OUT2, defOut1>} is invalid because is does not enable path continuity between IN1 and OUT1.

## 5.5 Responsibility definitions and dynamic responsibilities

Responsibilities are processing tasks (e.g. procedures, functions, actions, etc.) that are referenced by scenarios and by components. They are described in the responsibility-definitions element. A responsibility has a name, a description, and a unique identifier that can be referenced by components (in component-ref) and by paths (in the hyperedge responsibility-ref). Preconditions and post-conditions can be associated to a responsibility. A responsibility may also include a sequence of execution statements (*exec-sequence*) in a user-defined format. Such an execution sequence can be used for fine-grained modeling purpose, e.g. for performance analysis.

Responsibilities can be defined as being *dynamic*. Dynamic stubs show how behavior patterns can evolve at run time, whereas dynamic responsibilities and dynamic components show how a structure of components evolves at run time. A dynamic responsibility (dynamic-resp) performs an action on dynamic components or on plug-ins, in or out of a path. The different types of actions are:

- *create*: for dynamic components, creates a new component instance on a path (*in*) or in a slot or a pool of components (*out*). For plug-ins, this action creates a new plug-in instance on a path (*in*) or in a stub or a pool of plug-ins (*out*).
- *move*: for dynamic components, moves a component instance from a slot or a pool of components to a path (*in*) or from a path to a slot or a pool (*out*). For plug-ins, this action moves a plug-in instance from a stub or a pool of plug-ins to a path (*in*) or from a path to a stub or a pool (*out*).
- *move-stay*: similar to *move*, but moves a *reference* (i.e. an *alias*) to the component or plug-in while leaving the original instance in place. The same instance hence become visible in several places at once.
- *copy*: similar to *move*, but moves a distinct *copy* of to the component or plug-in while leaving the original instance in place. The two copies then evolve separately.
- *destroy*: for dynamic components, deletes a component instance found in a slot or a pool of components (*in*) or on a path (*out*). For plug-ins, this action deletes a plug-in instance found in a stub or a pool of plug-ins (*in*) or on a path (*out*).

A dynamic responsibility may also be associated to an explicit source pool. The *sourcepool* attribute is the *pool-name* of an existing pool in the fr-model. The relative length of the displayed arrow may also be specified (*arrow-length* is a real number between 0 and 1).

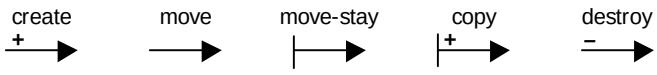
Responsibilities can also be used for describing performance annotations with the elements data-store-spec and service-request-spec. These elements will be discussed in Section 5.6

#### 5.5.1.1 XML definition

```
<!ELEMENT responsibility-definitions (responsibility)*>

<!ELEMENT responsibility (data-store-spec?, service-request-spec?,
                          (precondition-list?, postcondition-list?)?,
                          dynamic-resp?)?>

<!ATTLIST responsibility
                resp-id          ID      #REQUIRED
                resp-name        CDATA   #IMPLIED
                exec-sequence    CDATA   #IMPLIED
                description      CDATA   #IMPLIED >

<!ELEMENT dynamic-resp EMPTY>

<!ATTLIST dynamic-resp
                type         (move | move-stay | create
                               | copy | destroy)        #REQUIRED
                direction    (in | out)                #REQUIRED
                sourcepool   CDATA                     #IMPLIED
                arrow-length NMTOKEN                   #IMPLIED >
```

#### 5.5.1.2 Graphical notation

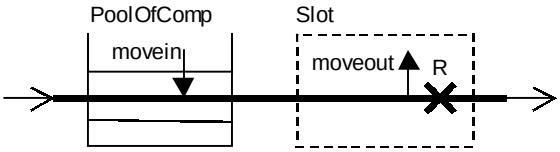
Responsibility definitions do not have a graphical presentation per se. However, references to these responsibilities are shown visually. Section 5.2.5 already presents the Graphical notation for references to non-dynamic responsibilities (i.e.



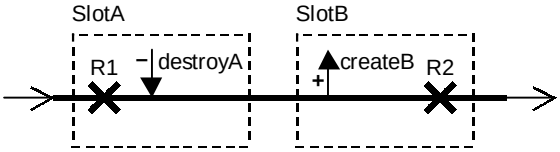

a cross on a path).

References to dynamic responsibilities are shown with arrows rather than with crosses. Different types of arrows are used for the five types of actions.

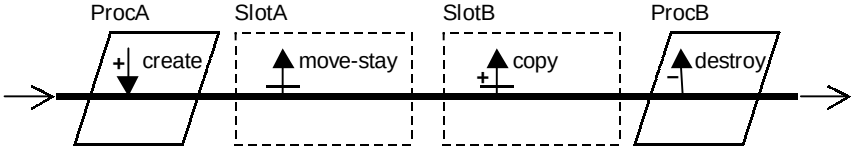
#### 5.5.1.3 Examples



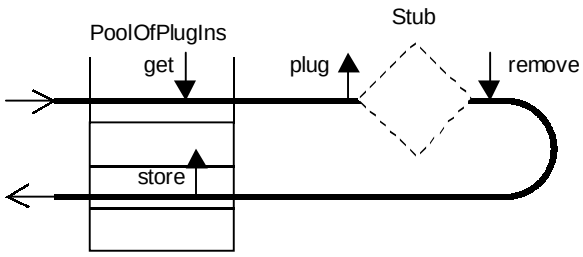

Dynamic responsibilities can be used to populate slots with dynamic components. In this example, a dynamic component is moved from a pool of components to the path (movein) and then moved out of the path to occupy a slot (moveout). This component can then perform responsibility R. This pool acts as a repository for inactive components.




The partial scenario shown above presupposes that SlotA is occupied by a dynamic component. After performing R1, this dynamic component is destroyed (destroyA), but the slot remains in order to be populated again at a later time. Then, a new dynamic component is created in SlotB (createB), which in turn performs R2.




In this third example, ProcA creates a new dynamic component and puts it on a scenario path. This dynamic component is placed in SlotA but at the same time remains on the path (just like a person can potentially occupy multiple roles at the same time in an organisation). A distinct copy is then placed in SlotB. Finally, ProcB destroys the original dynamic component by removing it from the path. This also means that SlotA is no longer occupied. However, SlotB is still populated by a copy of the original instance and is unaffected by the destroy action.




This last example illustrates how dynamic responsibilities can also be used to express mobile and dynamic behaviour. A plug-in is first taken out a pool of plug-ins (get) and then plugged in a dynamic stub (plug). The plug-in is executed (if the selection policy of the stub allows it). It is then removed from the stub (remove) and put back in the pool (store).

## 5.6 Annotations

Annotations represent a general mechanism by which structured information can be attached to models, components, and paths. The `annotations` element is part of a URN specification. In an `fr-model`, performance (response time) requirements and functional goals can be described in terms of these annotations. These categories of annotations are described in Sections 5.6.1 and 5.6.2 respectively.

### 5.6.1 Performance requirements

Performance requirements are composed of response-time requirements attached to scenario paths through timestamps. A `timestamp-point` is a `hyperedge` (Section 5.2) with a visual *`orientation`* attribute that describes, in degrees (0º for right, 90º for up, etc.), the orientation of the construct. A timestamp may also indicate whether it is used as a reference to the start of a performance requirement (*`next`*) or to its end (*`previous`*).

A response-time requirement (`response-time-req`) contains references to two timestamps (starting and ending timestamp points), a name, and a description. The attribute *`response-time`* indicates the required response time between the two timestamps (in µs) for a certain *`percentage`* of the responses.

Creating performance models from such requirements often necessitates the presence of an execution environment, e.g. devices and storage methods. Such an environment can be described at the beginning of a URN specification in the `nfr-definitions`. This element contains directories of devices and of data stores. A `device` has a name, a unique identifier, a description and a *`type`* (e.g. processor, disk, DSP, or other). A device may also have a predefined operation time (*`op-time`*). The `data-store-directory` contains data stores and access modes. `data-store` items and `access-mode` items have a unique identifier, a name and a description.

To enable performance analysis at the requirements level, responsibilities on a path can be linked to devices and can use data stores and access modes (see Section 5.5). Responsibility definitions have a `data-store-spec`, which defines how a data store is accessed through pairs of <reference to `data-store`, reference to `access-mode`>, and a `service-request-spec`, which contains a list of service requests. The latter are composed of a reference to a device (*`service-type`*) coupled with a user-defined quantity (*`request-number`*).

#### 5.6.1.1 XML definition

```
<!ELEMENT annotations (response-time-requirements?, agent-annotations?)>

<!ELEMENT response-time-requirements (response-time-req)*>

<!ELEMENT response-time-req EMPTY>

<!ATTLIST response-time-req
              timestamp1        IDREF   #REQUIRED
              timestamp2        IDREF   #REQUIRED
              resptime-name     CDATA   #IMPLIED
              response-time     NMTOKEN #REQUIRED
              percentage        NMTOKEN #REQUIRED
              description       CDATA   #IMPLIED >

<!ELEMENT timestamp-point EMPTY>

<!ATTLIST timestamp-point
              orientation       NMTOKEN            #IMPLIED
              reference         (previous | next)  #IMPLIED >

<!ELEMENT nfr-definitions (grl-concept-base, device-directory?,data-store-directory?)>

<!ELEMENT device-directory (device)*>

<!ELEMENT device EMPTY>

<!ATTLIST device
              device-id         ID                          #REQUIRED
              device-type       (processor|disk|dsp|other) #REQUIRED
              device-name       CDATA                       #IMPLIED
              description       CDATA                       #IMPLIED
              op-time           NMTOKEN                     #IMPLIED >

<!ELEMENT data-store-directory (data-store*, access-mode*)?>

<!ELEMENT data-store EMPTY>

<!ATTLIST data-store
              data-store-id     ID      #REQUIRED
              data-store-item   CDATA   #IMPLIED
              description       CDATA   #IMPLIED >

<!ELEMENT access-mode EMPTY>

<!ATTLIST access-mode
              access-mode-id    ID      #REQUIRED
              access-mode-item  CDATA   #IMPLIED
              description       CDATA   #IMPLIED >

<!ELEMENT data-store-spec (data-store-access)*>

<!ELEMENT data-store-access EMPTY>

<!ATTLIST data-store-access
              data-store-id     IDREF   #REQUIRED
              access-mode-id    IDREF   #REQUIRED >

<!ELEMENT service-request-spec (service-request)*>

<!ELEMENT service-request EMPTY>

<!ATTLIST service-request
              service-type      IDREF   #REQUIRED
              request-number    CDATA   #REQUIRED >
```

#### 5.6.1.2 Graphical notation


A timestamp point is shown as an arrowhead pointed towards the path where it is located. The *hyperedge-name* attribute of this hyperedge is shown as a label.

Because they are mostly definitions and annotations, the other elements presented in this section do not have a graphical annotation.

### 5.6.2 Functional goals

The second type of annotation considered in URN-FR relates to functional requirements goals in agent systems. An `fr-goal` contains two references to goal tags (starting and ending goal tags), preconditions and post-conditions, a unique identifier, a name, and a description. A `goal-tag` is similar in nature to a timestamp point: it is a `hyperedge` with an *`orientation`* attribute. However, the purpose of goal tags is to indicate on a path where explicit functional requirements goals are defined.

#### 5.6.2.1 XML definition

```
<!ELEMENT agent-annotations (fr-goal-list)>

<!ELEMENT fr-goal-list (fr-goal)*>

<!ELEMENT fr-goal (precondition-list?, postcondition-list?)>

<!ATTLIST fr-goal
               start-point  IDREF   #REQUIRED
               end-point    IDREF   #REQUIRED
               fr-goal-id   ID      #IMPLIED
               fr-goal-name CDATA   #IMPLIED
               description  CDATA   #IMPLIED >

<!ELEMENT goal-tag EMPTY>

<!ATTLIST goal-tag
               orientation NMTOKEN #IMPLIED >
```


#### 5.6.2.2 Graphical notation

A goal tag is shown as small square with a thick border located on a path. The *`hyperedge-name`* attribute of this hyperedge is shown as a label.

Because they are mostly annotations, the other elements presented in this section do not have a graphical annotation

## 5.7 Static semantic constraints and well-formedness rules

This section presents a set of static semantic constraints and well-formedness rules, related to identifier references and hyperedge connectivity that URN-FR specifications need to satisfy. Constraints on the structure of a URN specification already enforced by the URN DTD are not repeated in this section. Note that the following set of constraints is sound but not exhaustive: an unsatisfied constraint demonstrates that the URN-FR specification is not well formed, whereas some URN-FR specifications could be not well formed even when all these constraints are satisfied.

### 5.7.1 References to identifiers

In any XML DTD, attributes of type **`IDREF`** must refer to an existing attribute of type **`ID`**. Table 1 further refines this constraint by specifying the exact attribute referenced by attributes of type **`IDREF`**.

**Table 1** Constraints on referenced identifiers

| Element | IDREF Attribute | Referenced Element | Referenced ID Attribute | Additional Constraints |
|---|---|---|---|---|
| component-ref | *component-parent* | component-ref | *component-ref-id* | Both elements in the same structure-spec |
| | *referenced-component* | component | *component-id* | |
| data-store-access | *access-mode-id* | access-mode | *access-mode-id* | |
| | *data-store-id* | data-store | *data-store-id* | |
| enforce-bindings | *stub-id* | hyperedge | *hyperedge-id* | Both elements in the same hypergraph |
| fr-goal | *start-point* | hyperedge | *hyperedge-id* | hyperedge is a goal-tag |
| | *end-point* | hyperedge | *hyperedge-id* | hyperedge is a goal-tag |
| hyperedge-connection | *source-hyperedge* | hyperedge | *hyperedge-id* | Both elements in the same hypergraph |
| hyperedge-ref | *hyperedge-id* | hyperedge | *hyperedge-id* | Both elements in the same hypergraph |
| in-connection | *hyperedge-id* | hyperedge | *hyperedge-id* | hyperedge is a start from another fr-model in plug-in-maps |
| | *stub-entry-id* | stub-entry | *stub-entry-id* | |
| out-connection | *hyperedge-id* | hyperedge | *hyperedge-id* | hyperedge is an end-point from another fr-model in plug-in-maps |
| | *stub-exit-id* | stub-exit | *stub-exit-id* | |
| path-binding | *stub-entry-id* | stub-entry | *stub-entry-id* | stub-entry and stub-exit belong to same stub |
| | *stub-exit-id* | stub-exit | *stub-exit-id* | |
| path-branching-characteristic | *empty-segment* | hyperedge | *hyperedge-id* | hyperedge is an empty-segment |
| path-branching-spec | *fork-id* | hyperedge | *hyperedge-id* | hyperedge is a fork |
| plug-in-binding | *parent-map-id* | fr-model | *fr-model-id* | |
| | *stub-id* | hyperedge | *hyperedge-id* | hyperedge is a stub |
| | *submap-id* | fr-model | *fr-model-id* | fr-model is in plug-in-maps |
| plug-in-pool | *fr-model-id* | fr-model | *fr-model-id* | fr-model is in plug-in-maps |
| | *pool-id* | component | *component-id* | component is a pool |
| regular | *processor-id* | device | *device-id* | *device-type* is processor |
| response-time-req | *timestamp1* | hyperedge | *hyperedge-id* | hyperedge is a timestamp-point |
| | *timestamp2* | hyperedge | *hyperedge-id* | hyperedge is a timestamp-point |
| responsibility-ref | *resp-id* | responsibility | *resp-id* | |
| service-request | *service-type* | device | *device-id* | |
| stub-entry | *hyperedge-id* | hyperedge | *hyperedge-id* | hyperedge is an empty-segment |
| stub-exit | *hyperedge-id* | hyperedge | *hyperedge-id* | hyperedge is an empty-segment |

### 5.7.2 Well-formed rules for hypergraphs

Table 2 defines the constraints on how hyperedges can be connected together in order to form a well-formed hypergraph. For each type of hyperedge, the number of source/target hyperedges for normal path connections and for interpath

connections are specified, together with the types allowed. In general, `empty-segment` hyperedges are required between any two hyperedges of the other types, except for the `connect` and `abort` hyperedges. A cardinality of the form *x*-N means that the minimum is *x* and there is no maximum.

**Table 2** Constraints on hyperedge connectivity

| Hyperedge type | Normal path connections | | Interpath connections | |
|---|---|---|---|---|
| | # Sources | # Targets | # Sources | # Targets |
| `start` | 0 | 1 (empty-segment) | 0-N (connect) | 0 |
| `end-point` | 1 (empty-segment) | 0 | 0 | 0-N (connect) |
| `responsibility-ref` | 1 (empty-segment) | 1 (empty-segment) | 0 | 0 |
| `fork` | 1 (empty-segment) | 2-N (empty-segment) | 0 | 0 |
| `join` | 2-N (empty-segment) | 1 (empty-segment) | 0 | 0 |
| `synchronization` | 1-N (empty-segment) | 1-N (empty-segment) | 0 | 0 |
| `loop` | 2 (empty-segment) | 2 (empty-segment) | 0 | 0 |
| `stub` | 1-N (empty-segment) | 1-N (empty-segment) | 0-N (abort) | 0-N (abort) |
| `waiting-place` | 1 (empty-segment) | 1 (empty-segment), plus 0-1 (abort) for timeout path | 0-N (connect) | 0 |
| `abort` | 0 | 0 | 1 (stub or emtpy-segment) | 1 (stub or emtpy-segment) |
| | 1 (waiting-place/timer) | 1 (empty-segment) | 0 | 0 |
| `connect` | 0 | 0 | 1 (end-point or empty-segment) | 1 (start or waiting-place) |
| `timestamp-point` | 1 (empty-segment) | 1 (empty-segment) | 0 | 0 |
| `goal-tag` | 1 (empty-segment) | 1 (empty-segment) | 0 | 0 |
| `empty-segment` | 1 (any hyperedge except end-point and connect) | 1 (any hyperedge except start, abort and connect) | 0-N (abort) | 0-N (connect or abort) |

### 5.8 Dynamic Semantics

TO BE PROVIDED

Full-fledged formal languages often have a well-defined dynamic semantics. Whether a single dynamic semantics is desirable in a requirements language remains a question for further study. If one is defined, it should be done in a way that would enable simple transformations towards other languages such as UML, MSC, SDL, and TTCN.

## 6 Compliance statement

The purpose of this section is to state the requirements that the URN must fulfil and to what degree the notation does so.

### 6.1 Compliance table format

Table 3 presents the table format used to list each of the requirements defined for the URN (FR and NFR) notations and to assess whether the current definition of the notations complies, partially complies or does not comply with the requirement. If the notation complies (*C*), the standard must specify how it complies. If the notation does not comply (*N*), the standard must specify why it does not comply and what plans exist to bring the notation into compliance. If the notation partially complies (*P*), the standard must specify in what respects the notation complies and in what respects the notation does not comply and what plans exist to bring the notation into full compliance.

Each requirement possesses a unique identifier and is typed. A requirement is of type *FR* if it relates exclusively to functional requirements. A requirement is of type *NFR* if it relates exclusively to non-functional requirements. A requirement is of type *B* (both NFR and FR) if it relates to a relationship between functional and non-functional requirements. Requirements are also defined as being mandatory (*R*) or optional (*O*). A requirement may also depend on the presence of another requirement. Each requirement is cross-referenced to the sections where compliance to the requirement is discussed.

**Table 3** Compliance table format

| *ID* | *Requirement* | *Type* | *R/O* | *Depends On* | *Conf Status* | *Explanation* |
|---|---|---|---|---|---|---|
| 99998 | Specify data parameters for input events | FR | O | | P | Section *X*, element *E*. Data types not yet covered. |
| 99999 | Specify values for input events | FR | O | 99998 | N | Not specified |

In the example of Table 3, two optional (and artificial) requirements of type *FR* are provided. The first one is partially fulfilled by the notation. The second one depends on the presence of the first one, and the notation does not comply with it.

## 6.2 URN compliance table

TO BE PROVIDED

Table 4 lists each of the requirements defined for the URN (FR and NFR) and provides an assessment of compliance for the notations proposed in this document.

**Table 4** URN compliance table

| *ID* | *Requirement* | *Type* | *R/O* | *Depends On* | *Conf Status* | *Explanation* |
|---|---|---|---|---|---|---|
| 00100 | Specify the set of input events at scenario start point | FR | R | 5.2.4 | C | 5.2.2 |
| 00200 | Specify the set of output events at scenario end point | FR | R | 5.2.4 | C | 5.2.3 |
| 00300 | Specify preconditions at scenario start points | FR | R | 5.2.4 | C | 5.2.2 |
| 00400 | Specify post-conditions at scenario end points | FR | R | 5.2.4 | C | 5.2.3 |
| 00500 | Identify input sources, that is, whether the sources are human or machine | FR | R | | N | |
| 00600 | Identify output sources, that is, whether the sources are human or machine | FR | R | | N | |
| 00700 | Specify system operations in terms of a causal flow of responsibilities | FR | R | | C | 5.2 |
| 00800 | Specify alternative courses of action within a scenario | FR | R | | C | 5.2.6 |
| 00900 | Specify repetitive action within a scenario | FR | R | | C | 5.2.8 |
| 01000 | Specify parallel courses of action within a scenario | FR | R | | C | 5.2.7 |
| 01100 | Specify synchronization within a scenario | FR | R | | C | 5.2.7 |
| 01200 | Specify synchronization between scenarios | FR | R | | C | 5.2.10 |
| 01300 | Specify a lengthy scenario by way of a root map and references to child maps; child maps may have children | FR | R | 5.2.4, 5.4 | C | 5.1, 5.2.9 |
| 01301 | Specify preconditions at the entry points to a child map | FR | R | 01300, 5.2.4 | C | 5.2.9 |
| 01302 | Specify post-conditions at the exit points from a child map | FR | R | 01300, 5.2.4 | C | 5.2.9 |
| 01400 | Group related scenarios | FR | R | | C | 5.2 |
| 01500 | Specify feature interactions | FR | R | | C | 5.2 |
| 01600 | Specify scenarios without reference to components | FR | R | | C | 5.2 |

| *ID* | *Requirement* | *Type* | *R/O* | *Depends On* | *Conf Status* | *Explanation* |
|---|---|---|---|---|---|---|
| 01700 | Specify scenarios with reference to components and the allocation of responsibilities to components | FR | R | 5.2.1 | C | 5.3 |
| 01800 | Specify scenarios with reference to Commercial-Off-The-Shelf (COTS) components | FR | R | 01700 | C | |
| 01900 | Specify scenarios with reference to conceptual components | FR | R | 01700 | C | |
| 02000 | Specify the behaviour of the system's environment | FR | R | | C | Same requirements as for specifying scenarios |
| 02100 | Elicit requirements, that is, use the notation to reason about domain knowledge | FR | R | | C | |
| 02200 | Cross-reference operationalizations in the NFR model to responsibilities in the FR model | B | R | | N | |
| 02300 | Cross-reference performance constraints identified in the NFR model to responsibilities or scenarios in the FR model | B | R | | N | |
| 90100 | Specify ill-defined, tentative quality requirements | NFR | R | | C | |
| 90200 | Specify satisficing of quality requirements | NFR | R | | C | |
| 90300 | Specify refinement of quality requirements | NFR | R | | C | |
| 90400 | Specify alternative refinement of quality requirements | NFR | O | | C | |
| 90500 | Specify alternative functional requirements | NFR | R | | C | |
| 90600 | Specify quality requirement priorities | NFR | R | | P | Can be defined as Softgoal attributes. Might need to be part of the language. |
| 90700 | Specify synergies and conflicts among quality requirements | NFR | R | | C | |
| 90800 | Specify argumentation during modeling | NFR | R | | C | |
| 90900 | Specify multiple stakeholders' interests | NFR | R | | C | |
| 91000 | Specify business objectives | NFR | R | | C | |

| ID | Requirement | Type | R/O | Depends On | Conf Status | Explanation |
|---|---|---|---|---|---|---|
| 91100 | Specify links between high-level objectives and lower-level specifications | NFR | O | | C | |
| 91200 | Support requirements change traceability | NFR | R | | C | |
| 91300 | Support requirements priority traceability | NFR | R | | C | |
| 91400 | Integrate quality and functional requirements | B | R | | P | Further links between URN-FR & NFR could be provided |
| 91500 | Specify quantitative quality requirements | NFR | R | | P | Attributes provide partial support. Further support will be needed |
| 91600 | Support incremental commitments of requirements | NFR | R | | C | |
| 91700 | Knowledge base support | NFR | R | | N | Needs further work |
| 91800 | Support detection of conflicting and synergistic quality requirements | NFR | O | 91700 | N | Needs knowledge base and Correlation catalogue support |
| 91900 | Ease of use but also precision | NFR | R | | P | Further levels of formality need to be added |

# 7 Compliance testing

TO BE PROVIDED

The purpose of this section is to specify tests that can be applied to an implementation of the URN to determine its compliance with requirements.

# 8 Further study

The order of topics reflects importance.

## 8.1 Formal description of URN

The issues discussed under this heading relate to the formalization of the URN. How these issues are resolved will influence how the other issues requiring further study are resolved.

The first issue is how the URN-FR aligns with other SG10 notations such as MSC, SDL, and TTCN as well as with OMG's UML. The convergence of SDL with UML impacts how this issue is handled. One line of research could be to see if the main concepts behind URN-FR could be worked into the UML meta-model. The URN-DTD could by the same means be specified in a way that would conform to OMG's MOF and XMI standards. The core of the URN notation could be aligned with UML's, and detailed URN elements could be defined according to standard UML extension mechanisms (e.g. profiles). An evolution path towards UML 2.0 would be appropriate.

The second issue is how to add a formal data model to the URN-FR. Whatever data model is selected should be compatible with (translatable into) already standardized data models. One option may be to allow users to pick their preferred data model (ASN.1, ADT, URN-specific, user-defined, etc.).

The third issue relates to component types. Users could also define component types. The URN-FR contains a number of component types. Other possible sources for component type definitions are SDL, UML-RT, ADL, IN, or ODP component types.

The fourth issue concerns the formal connections between NFR and FR specifications. The URN-FR and URN-NFR notations specified in this standard grew up separately.

The fifth issue is the need to generalise the XML-based mechanism for describing URN-FR definitions and annotation so that it is extensible to NFR elements. Right now, they are tightly coupled to existing performance/goals annotations. In a context where various NFRs are considered, definitions and annotations would need to support all aspects of these NFRs.

The sixth issue is that a URN specification is now defined as a self-contained file. Mechanisms to support modularity, extensibility, and separate layout information are needed. Everything needs to be in one file.

## 8.2 Validation

The lack of formal ways to validate requirements specifications has meant that some issues are not discovered and resolved until the formal method of implementation has been applied. It has also meant that the release of requirements specifications has been delayed because informal review methods are used to accomplish validation, and these methods are less satisfactory at handling doubts and disagreements.

A functional requirements model created using a formally defined URN can be analysed using a formally defined algorithm. An issue is what properties should the algorithm be looking for? Since stakeholders can specify alternative courses of action using the URN-FR, they might want to know if a particular branch is ever taken and under what conditions. Determining branch coverage is conceivable assuming that execution semantics have been defined for the notation.

## 8.3 System data, system states, preconditions, post-conditions

The URN-FR allows stakeholders to express variations in system responses as a group of scenarios. For example, a call set-up attempt terminates differently depending on whether the destination terminal is already involved in a call or not. The stakeholder can use the URN-FR to express the alternate courses of action dependent on the destination terminal state using the OR-fork. The OR-fork can be annotated to express the dependency on the destination terminal state. Conceptually the URN-FR model for a call set-up attempt has embedded in it at least two scenarios.

The precondition for success is that the destination terminal is on-hook when the input event (the originator goes off-hook) is received and remains that way until ringing is applied to it as a result of this call attempt. The post-condition is ringing applied to the destination terminal and to the origination terminal.

The precondition for failure is that the destination terminal goes off-hook sometime prior to when this call attempt tests for the destination terminal state. The post-condition is that a busy signal is applied to the origination terminal.

The annotation mechanism in the URN-FR is incapable of handling complex mappings of preconditions to post-conditions associated with input events. A mechanism that is adequate to do so would assist in the generation of system test cases from the URN-FR model. See the section entitled “Requirements test case specification” for more discussion.

For further study is determining the desirability of allowing users to define variables for preconditions and post-conditions along with the ranges of values these variables can hold and allowing users to reference these variables in expressions attached to syntactical units containing conditions such as OR-forks and loops.

Such capability implemented in a tool would allow users to define a set of preconditions and then walk the URN-FR model for the input event to determine if the model successfully generates the post-conditions. If execution semantics were defined for the URN-FR, and if a tool implementing the URN-FR were available, a trace could be generated and used to verify the generation of the post-conditions.

Allowing for the declaration of variables and ranges of values for those variables entails allowing for assignments of values to variables. Values in a set of environmental variables could then be assigned to a set of system variables to form the preconditions for the execution of a scenario. The “execution” of responsibilities could result in system variables being assigned values, and these system variables could be referenced in loop and OR-fork conditions.

## 8.4 Executability

Executability means that a textual form of the URN-FR model can be compiled and run or that it can be interpreted. The Graphical notation for event triggering and termination represent inputs/outputs. The causality flow line represents sequential execution. The notation for branching and loops represent control constructs. The notation for synchronization represents operating system process constructs. What are missing are a formal data model, an expression evaluator, and assignment. If, in order to achieve executability, a responsibility requires some logic, it may be sufficient to replace the responsibility with a stub whose plug-in specifies the required logic and thus avoid defining a programming language for responsibilities. It would seem appropriate to attach assignment statements to responsibilities for transformations of internal system variables, to trigger points for the reception of environmental values, to termination points for the generation of output values, and to input and output ports on stubs.

Execution semantics would have to be defined for the following operations in order of complexity:

- sequence of assignments
- expression evaluation

- concurrency within one scenario (AND-forks)
- synchronization within one scenario (AND-joins)
- multiple, concurrent executions of the same scenario
- concurrent execution of different scenarios
- synchronization between different scenarios (waiting places)

## 8.5 Performance evaluation

Preliminary performance analysis can in principle be performed by building from the URN model either an analytic performance model or a simulation performance model. Both approaches depend on the existence of execution semantics for URN-FR models. The URN model generally will require amplification in a variety of ways to make a performance analysis possible. For example, some environment elements such as processors and networks must be included in the URN model, and some environment services such as file services must be defined.

The feasibility of doing performance analysis on the kind of preliminary specification defined by an URN-FR model has been demonstrated. However, further research is needed into the creation of different kinds of performance models from the particular data provided by URN-FR models. Some questions are:

- how to include environment services with complex behaviour
- probable interpretations of abstract URN-FR model elements
- efficient simulation
- automatic construction of suitable extended queueing models for analytic modelling

Other kinds of models may, in future, be constructed for other non-functional attributes such as reliability.

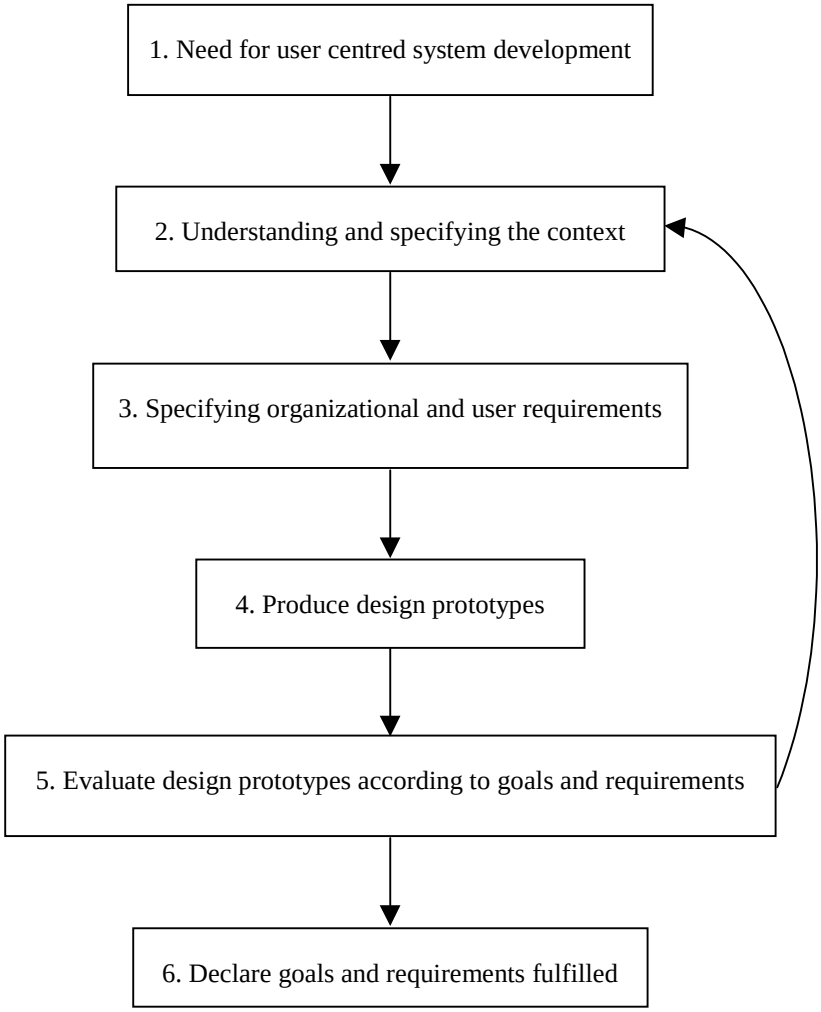
## 8.6 Usability

It is not enough that a toolmaker implement the URN as specified in the formal definition of the notation. There must be clear evidence that the usability of the implementation has been tested by trials conducted within the user community. How to do this is a matter for further study; see Appendix III.3 for a list of references that should help this study.

A requirement for a usability test is a set of use cases for the notations. A use case specification would consist of a set of instructions on how to create a particular URN-FR model and a particular URN-NFR model together with a specification of what the completed models look like and contain.

Another requirement for a usability test is a set of criteria for success so that the implementation can be declared to be usable.

Here is one high-level definition of a process for doing usability testing. Steps 1, 2, 3 and 6 take place within the context of creating this standard. The implementer does step 4. Step 5 is done by an independent agency.




## 8.7 Minor issues

### 8.7.1 Matching stubs with plug-ins

A plug-in may have more/fewer inputs/outputs than what is there on the stub and still satisfy the behavioural requirements of the scenario containing the stub. Consider the example of a stub with one input and one output and a plug-in with one input and two outputs. The second output of the plug-in may be to the environment. Or the plug-in has two inputs and one output. The two inputs are joined by an OR-fork, and the scenario preconditions guarantee that the second input does not occur. An assumption here is that the plug-ins satisfy the input to output transformation relations for the stub.

Another interesting example is a stub with two inputs and one output. A plug-in that joins the two inputs with an OR-fork to create a single output matches but then so does a plug-in that joins the two inputs with an AND-join to create a single output. The semantics, however, are very different. At the present time there is no way to specify the stub so that only one of these two plug-ins is allowed to match.

It would appear that some extension is required to stub semantics. A validation mechanism should help to determine whether a plug-in belongs in a particular stub.

### 8.7.2 Deeply nested maps

Deeply nested maps are the price paid for comprehensibility at any one map level and being able to specify large, complex systems. Given that maps generally contain more than one scenario specification, the user has to be able to navigate through the layers of nested maps in order to trace a particular scenario. A possible solution here is tool supported map navigation with animation driven by execution of the scenario.

### 8.7.3 Report generation

A goal of this standard is to define URN-FR so that it can be implemented in a tool. URN-FR, however, does not have graphical notations for all its elements. Some annotations, attributes or parameters may not be described visually. This standard must define a report format that presents together both graphical and non-graphical elements.

### 8.7.4 Graphical layout specification

At issue is whether graphical layout information is written to the output file generated by a tool that implements the notation.

# Annex A
# Document Type Definition

This annex presents a brief introduction to XML Document Type Definitions (DTDs) and it provides a graphical representation of the URN DTD elements. This annex also includes, for reference, a table cross-referencing elements and their attributes.

## A.1 Introduction to XML Document Type Definitions

XML DTDs describe the structure of a document. In this contribution, XML DTDs are composed of *elements*, which describe the markups or tags in the URN language, and of *attributes*, which represent parameters associated to a specific element.

An element is of the form:

```
<!ELEMENT element-name (sub-elements or EMPTY)>
```

An *empty* element indicates a terminal element that does not contain any sub-element. *Sub-elements* can be structured in various ways:

- As an ordered sequence: `(element1, element2, element3)`
- As a choice among elements: `(element1 | element2 | element3)`
- As an optional element: `(element?)`
- As multiple instances of an element (0 to N): `(element*)`
- As multiple instances of an element (1 to N): `(element+)`
- As a combination of the options cited above: `( (element1?, (element2* | element3+) )`

A list of attributes can be associated to the definition of an element. An attribute is of the form:

```
<!ATTLIST element-name
                  attribute-name1     type   value
                  attribute-name2     type   value
                  attribute-name3     type   value
                  ...                              >
```

The *type* of an attribute is one of the following:

- **`ID`**: Unique identifier value.
- **`IDREF`**: Reference to a unique identifier value.
- **`NMTOKEN`**: Name toked (one word).
- **`CDATA`**: A string of characters, with spaces allowed.
- `(name1 | name2 | name3)`: Group of names (enumeration).

The *value* of an attribute is one of the following:

- **`#REQUIRED`**: A value must be provided for this attribute.
- **`#IMPLIED`**: A value may be provided for this attribute; otherwise an implicit value is given.
- `name1`: A default item in a group of names.
- `default`: A default name token or string of character data.

As an example, assume the following DTD:

```
<!ELEMENT family (father, mother, child*)>
<!ATTLIST family
                  family-name   CDATA  #IMPLIED >

<!ELEMENT father (EMPTY)>
<!ATTLIST father
                  name          CDATA  #IMPLIED >

<!ELEMENT mother (EMPTY)>
<!ATTLIST mother
                  name          CDATA  #IMPLIED >

<!ELEMENT child (EMPTY)>
<!ATTLIST child
                  name          CDATA  #IMPLIED
                  sex           (male | female) "female" >
```

The following XML document is *valid* according to this DTD:

```
<family family-name="Smith">
     <father name="John" />
     <mother name="Mary Jane" />
     <child name="Anna" />
     <child name="Martin" sex="male"></child>
</family>
```

## A.2 Graphical representation of URN DTD elements

The conventions presented in Figure 15 are used to describe the URN DTD elements in a graphical form.

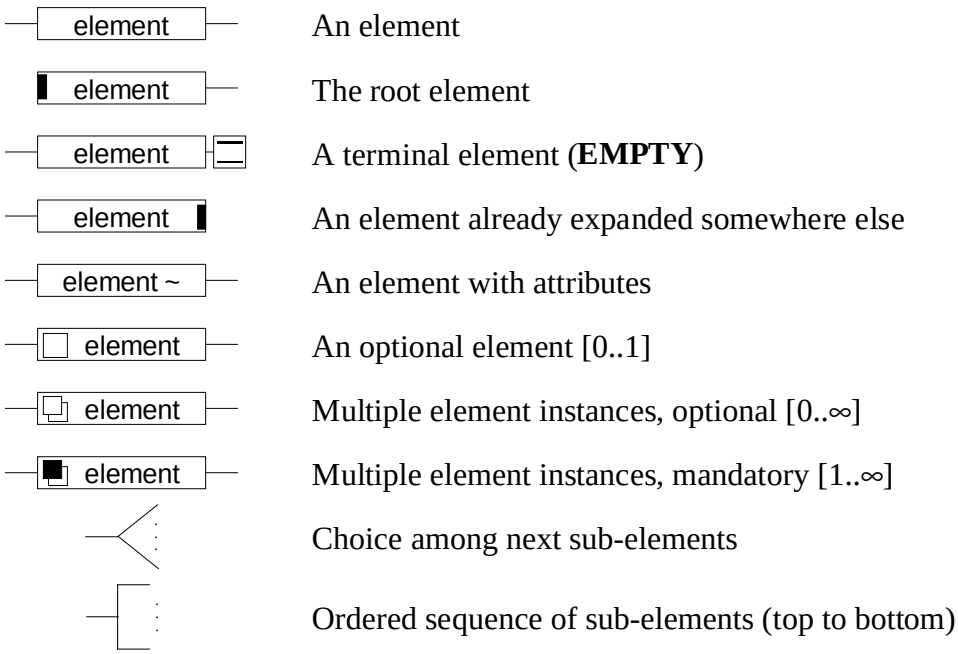



**Figure 15.** Graphical conventions for DTD elements

Due to the large size of the URN DTD, its graphical representation will be split into four parts: definitions, urn-nfr-spec, urn-fr-spec, and annotations.

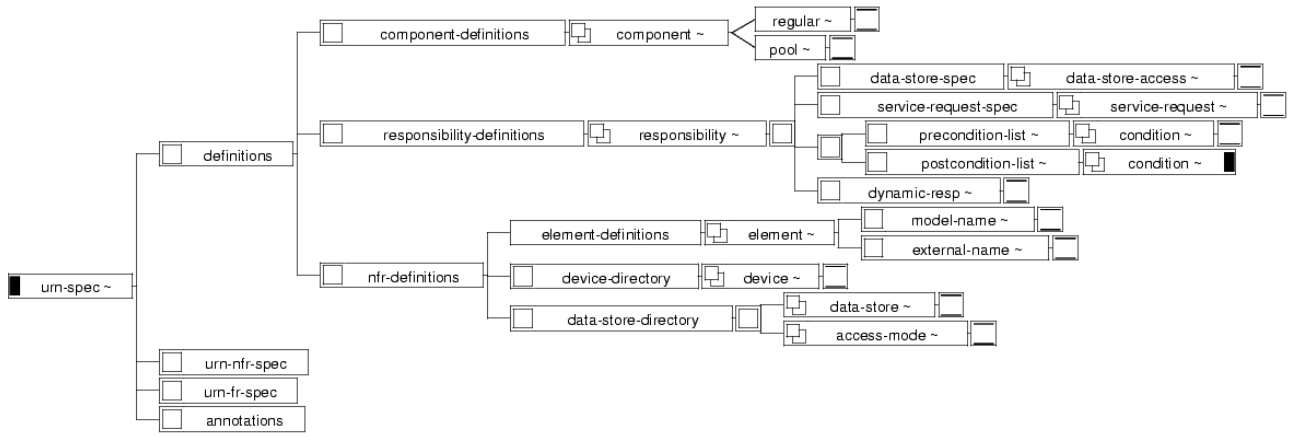



**Figure 16.** Sub-elements of definitions

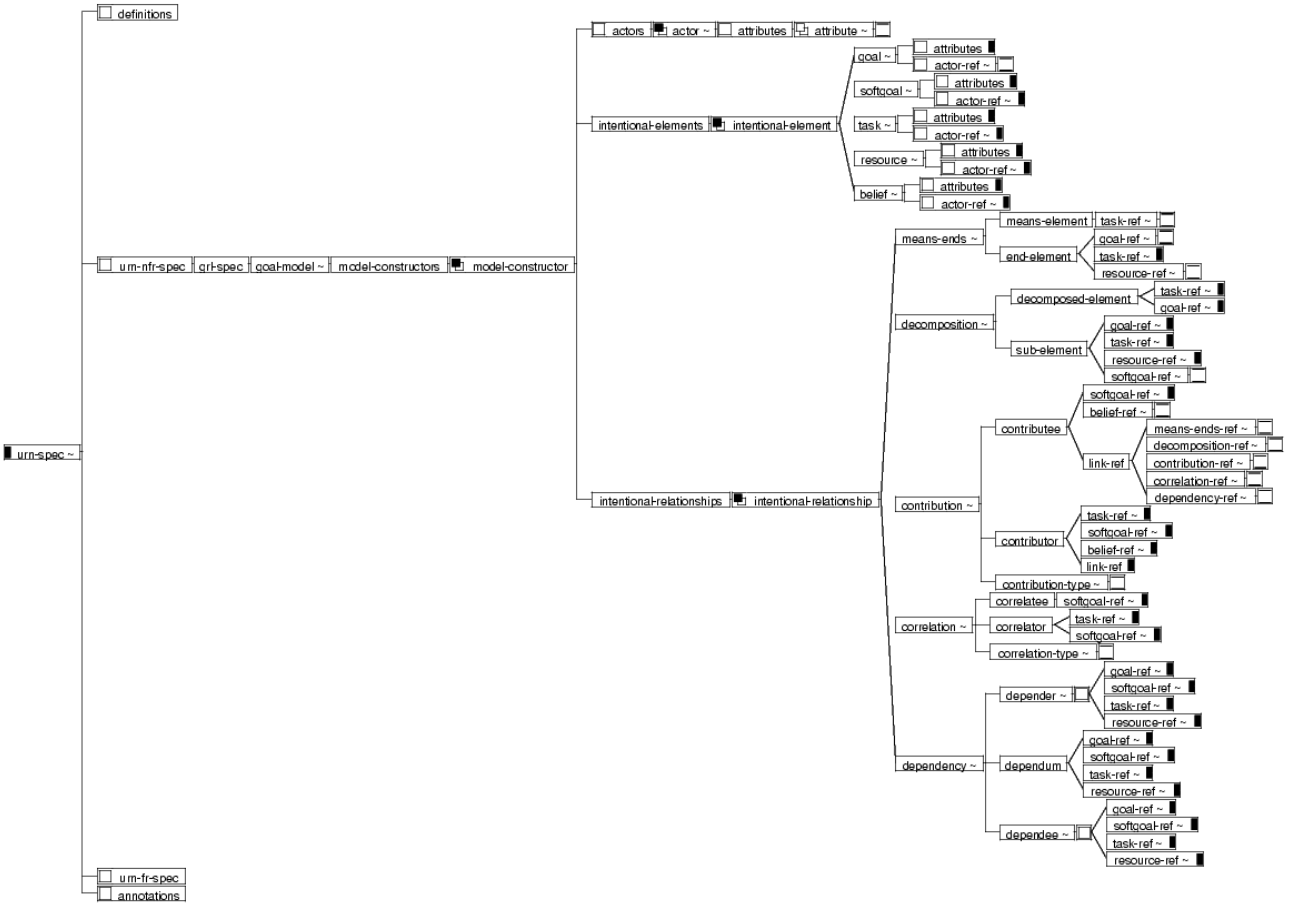



**Figure 17.** Sub-elements of urn-nfr-spec

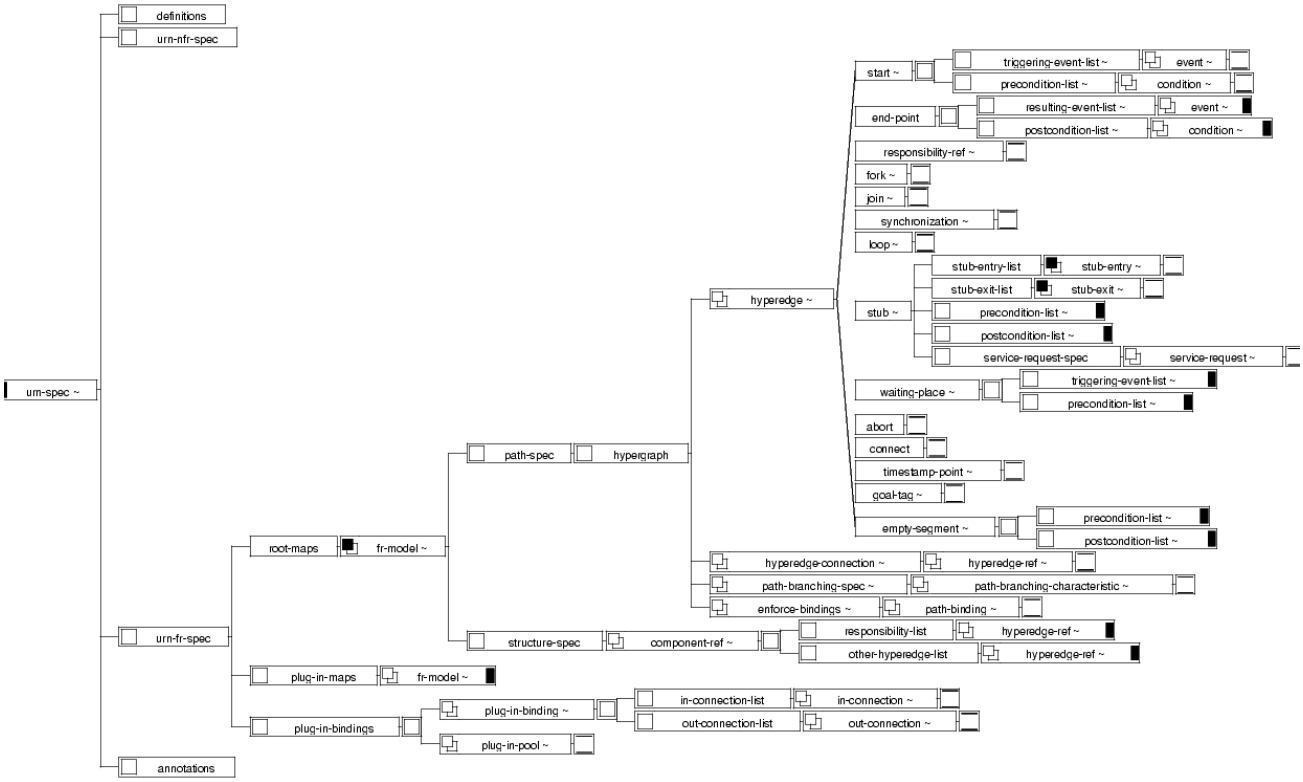



**Figure 18.** Sub-elements of urn-fr-spec

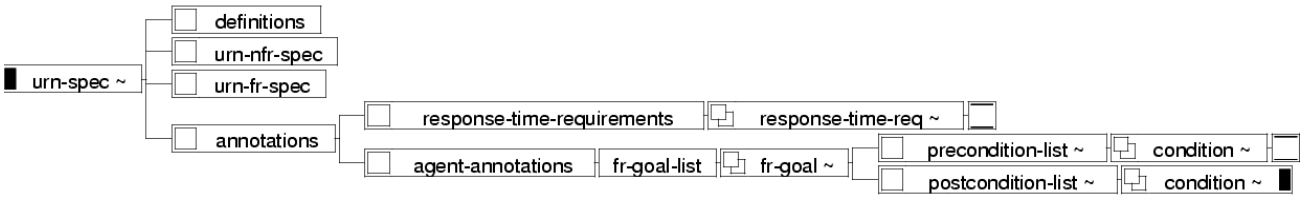



**Figure 19.** Sub-elements of annotations

## A.3 URN elements and their attributes

The following table summarizes the attributes associated to each element in the URN DTD. It is provided here to complement the information missing from the graphical representation presented in the previous section.

**Table 5** Sorted list of elements and their attributes.

| | **Attribute** | | | |
|---|---|---|---|---|
| **Element Name** | **Name** | **Type** | **Allowable Values** | **Default** |
| **access-mode** | *access-mode-id* | Identifier Value | | REQUIRED |
| | *access-mode-item* | Character Data | | IMPLIED |
| | *description* | Character Data | | IMPLIED |
| **actor** | *actor-id* | Identifier Value | | REQUIRED |
| | *actor-name* | Character Data | | IMPLIED |
| | *description* | Character Data | | IMPLIED |
| **actor-ref** | *actor-id-ref* | Identifier Ref | | REQUIRED |
| **attribute** | *element-id-ref* | Identifier Ref | | REQUIRED |
| **belief** | *belief-id* | Identifier Value | | REQUIRED |
| | *belief-name* | Character Data | | IMPLIED |
| | *description* | Character Data | | IMPLIED |

| | | | | |
|---|---|---|---|---|
| **belief-ref** | *belief-id-ref* | Identifier Ref | | REQUIRED |
| **component** | *colour* | Name Token | | IMPLIED |
| | *component-id* | Identifier Value | | REQUIRED |
| | *component-name* | Character Data | | REQUIRED |
| | *description* | Character Data | | IMPLIED |
| **component-ref** | *anchored* | Group of Names | yes,no | no |
| | *component-parent* | Identifier Ref | | IMPLIED |
| | *component-ref-id* | Identifier Value | | REQUIRED |
| | *fx* | Name Token | | REQUIRED |
| | *fy* | Name Token | | REQUIRED |
| | *height* | Name Token | | REQUIRED |
| | *referenced-component* | Identifier Ref | | IMPLIED |
| | *role* | Character Data | | IMPLIED |
| | *width* | Name Token | | REQUIRED |
| **condition** | *description* | Character Data | | IMPLIED |
| | *name* | Name Token | | REQUIRED |
| **contribution** | *contribution-id* | Identifier Value | | REQUIRED |
| **contribution-ref** | *contribution-link-id-ref* | Identifier Ref | | REQUIRED |
| **contribution-type** | *contri-type* | Group of Names | break,hurt,some-negative, unknown, equal, some-positive, help,make,and,or | REQUIRED |
| **correlation** | *correlation-id* | Identifier Value | | REQUIRED |
| **correlation-ref** | *correlation-link-id-ref* | Identifier Ref | | REQUIRED |
| **correlation-type** | *correlation-type* | Group of Names | break,hurt,some-negative, unknown, some-positive, help, make,equal | REQUIRED |
| **data-store** | *data-store-id* | Identifier Value | | REQUIRED |
| | *data-store-item* | Character Data | | IMPLIED |
| | *description* | Character Data | | IMPLIED |
| **data-store-access** | *access-mode-id* | Identifier Ref | | REQUIRED |
| | *data-store-id* | Identifier Ref | | REQUIRED |
| **decomposition** | *decomposition-id* | Identifier Value | | REQUIRED |
| **decomposition-ref** | *decomposition-id-ref* | Identifier Ref | | REQUIRED |
| **dependee** | *actor-id-ref* | Identifier Ref | | REQUIRED |
| **dependency** | *dependency-id* | Identifier Value | | REQUIRED |
| **dependency-ref** | *dependency-id-ref* | Identifier Ref | | REQUIRED |
| **depender** | *actor-id-ref* | Identifier Ref | | REQUIRED |
| **device** | *description* | Character Data | | IMPLIED |
| | *device-id* | Identifier Value | | REQUIRED |
| | *device-name* | Character Data | | IMPLIED |
| | *device-type* | Group of Names | processor,disk,dsp,other | REQUIRED |
| | *op-time* | Name Token | | IMPLIED |
| **dynamic-resp** | *arrow-length* | Name Token | | IMPLIED |
| | *direction* | Group of Names | in,out | REQUIRED |
| | *sourcepool* | Character Data | | IMPLIED |
| | *type* | Group of Names | move,move-stay,create,copy, destroy | REQUIRED |
| **element** | *description* | Character Data | | IMPLIED |
| | *element-id* | Identifier Value | | REQUIRED |
| | *name* | Character Data | | IMPLIED |

| | | | | |
|---|---|---|---|---|
| **empty-segment** | *characteristics* | Group of Names | failure-point,shared,direction-arrow | IMPLIED |
| | *path-label* | Character Data | | REQUIRED |
| | *show-label* | Group of Names | yes,no | no |
| **enforce-bindings** | *stub-id* | Identifier Ref | | REQUIRED |
| **event** | *description* | Character Data | | IMPLIED |
| | *name* | Name Token | | REQUIRED |
| **external-name** | *description* | Character Data | | IMPLIED |
| | *id* | Identifier Value | | REQUIRED |
| | *name* | Character Data | | IMPLIED |
| **fork** | *orientation* | Name Token | | IMPLIED |
| **fr-goal** | *description* | Character Data | | IMPLIED |
| | *end-point* | Identifier Ref | | REQUIRED |
| | *fr-goal-id* | Identifier Value | | IMPLIED |
| | *fr-goal-name* | Character Data | | IMPLIED |
| | *start-point* | Identifier Ref | | REQUIRED |
| **fr-model** | *description* | Character Data | | IMPLIED |
| | *fr-model-id* | Identifier Value | | REQUIRED |
| | *fr-model-name* | Character Data | | REQUIRED |
| | *title* | Character Data | | No title |
| **goal** | *description* | Character Data | | IMPLIED |
| | *goal-id* | Identifier Value | | REQUIRED |
| | *goal-name* | Character Data | | IMPLIED |
| **goal-model** | *description* | Character Data | | IMPLIED |
| | *goal-model-id* | Identifier Value | | REQUIRED |
| | *goal-model-name* | Character Data | | IMPLIED |
| **goal-ref** | *goal-id-ref* | Identifier Ref | | REQUIRED |
| **goal-tag** | *orientation* | Name Token | | IMPLIED |
| **hyperedge** | *description* | Character Data | | IMPLIED |
| | *fx* | Name Token | | IMPLIED |
| | *fy* | Name Token | | IMPLIED |
| | *hyperedge-id* | Identifier Value | | REQUIRED |
| | *hyperedge-name* | Character Data | | IMPLIED |
| | *lx* | Name Token | | IMPLIED |
| | *ly* | Name Token | | IMPLIED |
| **hyperedge-connection** | *source-hyperedge* | Identifier Ref | | REQUIRED |
| **hyperedge-ref** | *hyperedge-id* | Identifier Ref | | REQUIRED |
| **in-connection** | *hyperedge-id* | Identifier Ref | | REQUIRED |
| | *stub-entry-id* | Identifier Ref | | REQUIRED |
| **join** | *orientation* | Name Token | | IMPLIED |
| **loop** | *exit-condition* | Character Data | | IMPLIED |
| | *orientation* | Name Token | | REQUIRED |
| **means-ends** | *means-ends-id* | Identifier Value | | REQUIRED |
| **means-ends-ref** | *means-ends-id-ref* | Identifier Ref | | REQUIRED |
| **model-name** | *description* | Character Data | | IMPLIED |
| | *model-id* | Identifier Value | | REQUIRED |
| | *name* | Character Data | | IMPLIED |
| **out-connection** | *hyperedge-id* | Identifier Ref | | REQUIRED |
| | *stub-exit-id* | Identifier Ref | | REQUIRED |

| | | | | |
|---|---|---|---|---|
| **path-binding** | *stub-entry-id* | Identifier Ref | | REQUIRED |
| | *stub-exit-id* | Identifier Ref | | REQUIRED |
| **path-branching-characteristic** | *characteristic* | Character Data | | IMPLIED |
| | *empty-segment* | Identifier Ref | | REQUIRED |
| | *lx* | Name Token | | IMPLIED |
| | *ly* | Name Token | | IMPLIED |
| | *probability* | Name Token | | IMPLIED |
| **path-branching-spec** | *fork-id* | Identifier Ref | | REQUIRED |
| **plug-in-binding** | *parent-map-id* | Identifier Ref | | REQUIRED |
| | *stub-id* | Identifier Ref | | REQUIRED |
| | *submap-id* | Identifier Ref | | REQUIRED |
| **plug-in-pool** | *fr-model-id* | Identifier Ref | | REQUIRED |
| | *pool-id* | Identifier Ref | | REQUIRED |
| **pool** | *type* | Name Token | | REQUIRED |
| **postcondition-list** | *composition* | Character Data | | AND |
| **precondition-list** | *composition* | Character Data | | AND |
| **regular** | *processor-id* | Identifier Ref | | IMPLIED |
| | *protected* | Group of Names | yes,no | no |
| | *replicated* | Group of Names | yes,no | no |
| | *replication-factor* | Name Token | | 1 |
| | *slot* | Group of Names | yes,no | no |
| | *type* | Name Token | | team |
| **resource** | *description* | Character Data | | IMPLIED |
| | *resource-id* | Identifier Value | | REQUIRED |
| | *resource-name* | Character Data | | IMPLIED |
| **resource-ref** | *resource-id-ref* | Identifier Ref | | REQUIRED |
| **response-time-req** | *description* | Character Data | | IMPLIED |
| | *percentage* | Name Token | | REQUIRED |
| | *response-time* | Name Token | | REQUIRED |
| | *resptime-name* | Character Data | | IMPLIED |
| | *timestamp1* | Identifier Ref | | REQUIRED |
| | *timestamp2* | Identifier Ref | | REQUIRED |
| **responsibility** | *description* | Character Data | | IMPLIED |
| | *exec-sequence* | Character Data | | IMPLIED |
| | *resp-id* | Identifier Value | | REQUIRED |
| | *resp-name* | Character Data | | IMPLIED |
| **responsibility-ref** | *arrow-position* | Name Token | | IMPLIED |
| | *resp-id* | Identifier Ref | | REQUIRED |
| **resulting-event-list** | *composition* | Character Data | | AND |
| **service-request** | *request-number* | Character Data | | REQUIRED |
| | *service-type* | Identifier Ref | | REQUIRED |
| **softgoal** | *description* | Character Data | | IMPLIED |
| | *softgoal-id* | Identifier Value | | REQUIRED |
| | *softgoal-name* | Character Data | | IMPLIED |
| **softgoal-ref** | *softgoal-id-ref* | Identifier Ref | | REQUIRED |

| | | | | |
|---|---|---|---|---|
| **start** | *arrival* | Group of Names | exponential,deterministic,uniform, erlang, expert, none | none |
| | *expert-distribution* | Character Data | | IMPLIED |
| | *high* | Name Token | | IMPLIED |
| | *kernel* | Name Token | | IMPLIED |
| | *low* | Name Token | | IMPLIED |
| | *mean* | Name Token | | IMPLIED |
| | *value* | Name Token | | IMPLIED |
| **stub** | *selection-policy* | Character Data | | IMPLIED |
| | *shared* | Group of Names | yes,no | no |
| | *type* | Group of Names | static, dynamic | static |
| **stub-entry** | *hyperedge-id* | Identifier Ref | | REQUIRED |
| | *stub-entry-id* | Identifier Value | | REQUIRED |
| **stub-exit** | *hyperedge-id* | Identifier Ref | | REQUIRED |
| | *stub-exit-id* | Identifier Value | | REQUIRED |
| **synchronization** | *cardinality-source* | Name Token | | IMPLIED |
| | *cardinality-target* | Name Token | | IMPLIED |
| | *orientation* | Name Token | | IMPLIED |
| **task** | *description* | Character Data | | IMPLIED |
| | *task-id* | Identifier Value | | REQUIRED |
| | *task-name* | Character Data | | IMPLIED |
| **task-ref** | *task-id-ref* | Identifier Ref | | REQUIRED |
| **timestamp-point** | *orientation* | Name Token | | IMPLIED |
| | *reference* | Group of Names | previous, next | IMPLIED |
| **triggering-event-list** | *composition* | Character Data | | OR |
| **urn-spec** | *component-notation* | Name Token | | Buhr-UCM |
| | *data-language* | Name Token | | none |
| | *description* | Character Data | | IMPLIED |
| | *dtd-version* | Name Token | | REQUIRED |
| | *height* | Name Token | | 1100 |
| | *spec-id* | Identifier Value | | REQUIRED |
| | *spec-name* | Character Data | | IMPLIED |
| | *width* | Name Token | | 1320 |
| **waiting-place** | *timer* | Group of Names | yes,no | no |
| | *wait-type* | Character Data | | IMPLIED |

# Annex B
# Tutorial

This informative annex contains tutorials and illustrative examples that supplement the specifications of the URN-NFR and URN-FR languages (Sections 4 and 5).

## B.1 URN-NFR tutorial

This section provides a tutorial to show how the GRL language can be used to analyse Non-Functional Requirement within an application domain. We will first give a brief description of the example application domain and then use GRL to describe how NFRs are dealt with in this domain. Note that this example provides for one particular way GRL can be used, which emphasizes softgoals during modelling. There are other ways GRL can be used such as placing emphasis on actors, or on task and goals during the modelling.

### B.1.1 Application Description

An information system at Financial Institution provides support for point of sale systems for financial transactions. Security issues in such system concerns both the system developer and the system user. The remote input systems are located at Retailers sites (in the “field” outside of the financial institution itself) and are used by the Retailers to allow their customers to pay for their purchases. Other pertinent financial services are provided to the Retailers that can be done through the system. The system itself consists of a Base Station (BS) and a set of Terminals, and a Host computer. Terminals are connected to the Base Station via a LAN, while the Base Station in connected through a leased or dial-up line to the host computer located at the financial institutions. A software development contractor develops the financial transaction software that resides at the Retailers and on the Host computer. The hardware is purchased externally from a Manufacturer. There are two "business processes" relevant for this example: 1. When the system is newly deployed. 2. When Terminal software need to be updated.

Case 1: When the system is newly deployed:

The developers receive hardware from the manufacturer, and load the initial version of the software into the system. The system is then provided to the financial institution where it is validated. The financial institution then provides the system to retailers and keeps track of what software and hardware was provided to whom, and other pertinent information relevant to the Retailer (such as special configuration data). Finally, the Retailer operates the system.

Case 2: When the system is updated:

Developers provide updates to the financial institution. The Financial institution uploads the updates to their host computer. Maintainers in the field download the updates into the base station, from where they then transfer it to the terminals.

There is a need to address security needs when producing, deploying and updating the financial software at the financial institution and at the Retailers site. The basic problem to be addressed is, how to keep the software code secure both in its source code form during development, and when the software is deployed as object code, and related parameter tables. For example, how to ensure that only authorised personnel can upload new updates to the Host, download updates from the Host and install them onto the base station and terminals, how to ensure that only authorised terminals are used. These and other security issues need to be addressed during the specification and development of the financial software itself and the software that facilitate the maintenance processes. Security touches on ease of use, performance, and cost; these attributes need to be considered and traded off among each other.

### B.1.2 GRL Definitions

We will start by defining the existing modelling elements imported from an external model in some modelling notation (e.g. UML):

**ELEMENT** Software

**ELEMENT** TerminalSoftware

**ELEMENT** BaseStationSoftware

**ELEMENT** HostComputerSoftware

These statements provide a definition of the kind of elements that need to be secured. These elements were imported from an external model.

The following statement defines a softgoal on the security of the software of the system:

**SOFTGOAL** Security **OF** Software

where “Security” is a softgoal type, and “Software” is a softgoal topic. For the purpose of referencing this softgoal, a default name is automatically generated, which is the concatenation of the softgoal type, “OF” and the topic, e.g. “SecurityOFSoftware”. The user can override the default name with this alternative form of the statement:

**SOFTGOAL** SoftwareSecured **IS** Security **OF** Software

Let us now refine that softgoal to more specific ones. Refining the softgoal clarifies what we mean by Security of Software, and it may be achieved.

**SOFTGOAL** OperationalSecurity **OF** Software

**SOFTGOAL** DevelopmentalSecurity **OF** Software

**CONTRIBUTION**

OperationalSecurityOFSoftware,

DevelopmentalSecurityOFSoftware

**HAS** And **CONTRIBUTION-TO** SecurityOFSoftware


These statements refine what “Security OF Software” means. It means security of software during operational activities, that is all activities related to the provision, updating, and regular working of the software, and it means security of software during development activities. The contribution statement also states that both (“And”) are needed in order to ensure security of software. Security is not dealt with well when having stringent security for operations, while having “free” access to the source code during development.

Next, we recognise that for operational security, one should distinguish between operations that take place within financial institution and the development site, and operations that take place outside of these boundaries, such as at the Retailers site. This distinction can be shown as follows:

**SOFTGOAL** InternalOperationalSecurity **OF** Software

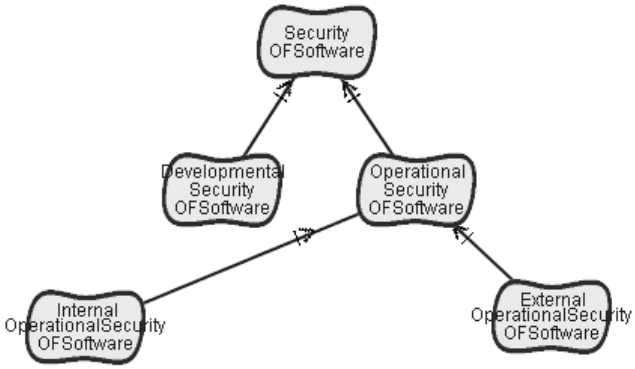
**SOFTGOAL** ExternalOperationalSecurity **OF** Software

**CONTRIBUTION**

InternalOperationSecurityOFSoftware,

ExternalOperationalSecurityOFSoftware

**HAS** And **CONTRIBUTION-TO** OperationalSecurityOFSoftware



We now also wish to distinguish among security that is related to the securing of the Terminal and of the Base Station and security measures related to the host computer that might be attached through activities performed externally. These considerations are expressed by the following definitions:

**SOFTGOAL** ExternalOperationalSecurity **OF** TerminalSoftware

**SOFTGOAL** ExternalOperationalSecurity **OF** BaseStationSoftware

**SOFTGOAL** ExternalOperationalSecurity **OF** HostComputerSoftware

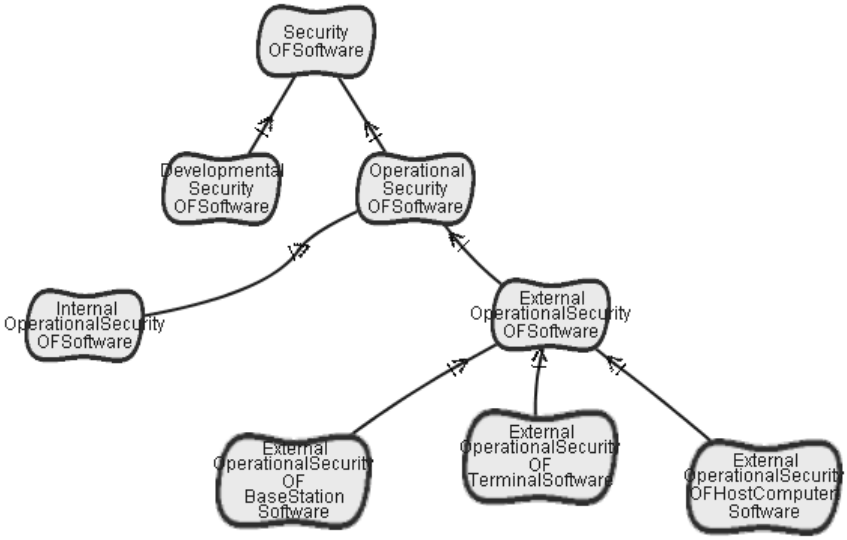
**CONTRIBUTION**

ExternalOperationalSecurityOFTerminalSoftware,

ExternalOperationalSecurityOFBaseStationSoftware,

ExternalOperationalSecurityOFHostComputerSoftware

**HAS** And **CONTRIBUTION-TO** ExternalOperationalSecurityOFSoftware




We could now make another distinction between securing application related software and operating system software, if we wish to deal with them differently. Perhaps operating system software for the pieces of equipment needs at least as stringent security measures as the financial applications running on top of them. However, we will not introduce this distinction here.

Let us now further focus on the TerminalSoftware and refine security into confidentiality, integrity and Availability. Confidentiality is protection against unauthorized disclosure. Integrity is protection against unauthorized update or tampering, and availability is protection against interruption of service through outside attacks. This refinement makes the meaning of security more particular in that we know what aspects of security we would further like to address. This refinement is expressed by the following definitions:

**SOFTGOAL** ExternalOperationalConfidentiality OF TerminalSoftware

**SOFTGOAL** ExternalOperationalIntegrity OF TerminalSoftware

**SOFTGOAL** ExternalOperationalAvailability OF TerminalSoftware

**CONTRIBUTION**

ExternalOperationalConfidentialityOFTerminalSoftware,

ExternalOperationalIntegrityOFTerminalSoftware,

ExternalOperationalAvailabilityOFTerminalSoftware

**HAS** And **CONTRIBUTION-TO** ExternalOperationalSecurityOFTerminalSoftware

We will now focus on the operation of update, download and storage provided by the terminal software for which we wish to have confidentiality ensured. This is shown by the following statements. Note that in order to shorten the names of the softgoal we will override the internally generated name. We will use the acronym EOC standing for "ExternalOperationalConfidentiality":

**SOFTGOAL** EOC_OFTerminalSoftwareForStorage **IS**

ExternalOperationalConfidentialityForStorage **OF** TerminalSoftware

**SOFTGOAL** EOC_OFTerminalSoftwareForDownload **IS**

ExternalOperationalConfidentialityForDownload **OF** TerminalSoftware

**SOFTGOAL** EOC_OFTerminalSoftwareForUpdate **IS**

ExternalOperationalConfidentialityForUpdate **OF** TerminalSoftware

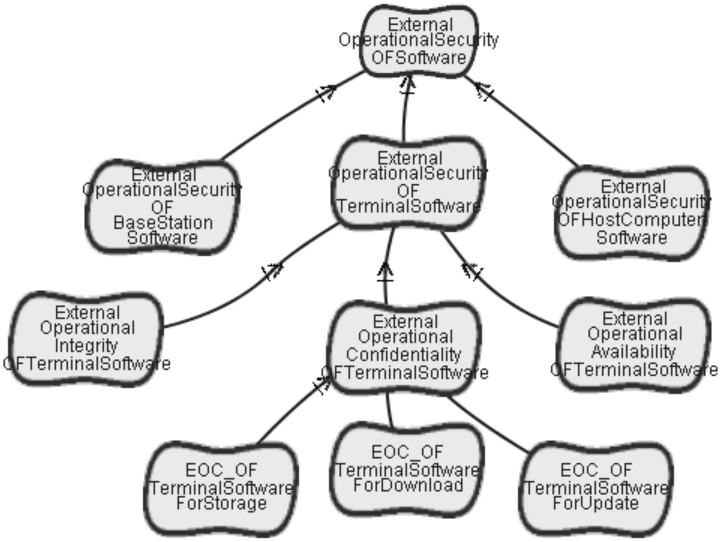
**CONTRIBUTION**

EOC_OFTerminalSoftwareForStorage,

EOC_OFTerminalSoftwareForDownload,

EOC_OFTerminalSoftwareForUpdate

**HAS** And **CONTRIBUTION-TO** EOC_OFTerminalSoftware




Up until now we have only provided refinements for making the softgoal of having secure software more precise, and found that, among other things, it means to find a way to have storage, download and update of terminal software made confidential, i.e. have the software in now way disclosed to unauthorised parties. We will now focus on the various means we could employ for achieving these goals. Let us now focus on the different means we can employ for achieving confidentiality for the updates of terminal software. Three means are discussed: providing access authorisation, providing some kind of encryption, and providing limited exposure to accessing the software. Each one of these means does have additional alternatives. Let us model one at the time:

**TASK** AccessAuthorization

**TASK** LimitExposure

**TASK** Encryption

**CONTRIBUTION**

AccessAuthorization,

LimitExpose,

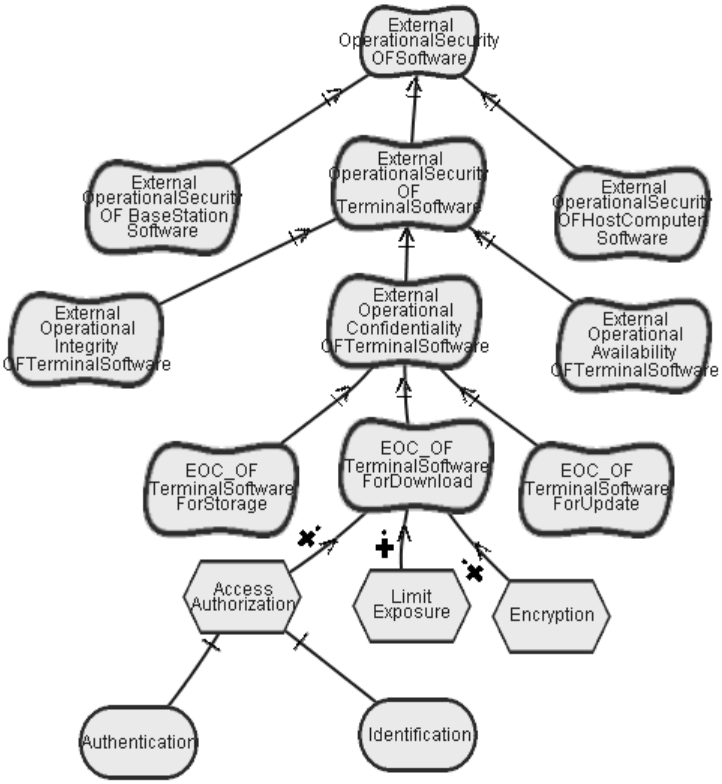
Encryption

**HAS** Make **CONTRIBUTION-TO** EOC_OFTerminalSoftwareForDownload

The Task AccessAuthorization has two subtasks: Authentication and Identification while each subtask provides a few alternatives ways. Since both of these tasks provide alternatives we will model them using the GOAL construct, and then provide for each alternative a task together with a means end link to its goal.

**GOAL** Authentication

**GOAL** Identification




Authentication can be done by using digital signatures, biometrics, card key and card reader equipment or password protection. Let us describe those alternative uses.

**TASK** DigitalSignatureAuthenication

**TASK** BiometricsAuthenication

**TASK** CardkeyAuthenication

**TASK** PasswordAuthentication

**MEANS-END FROM** DigitalSignature **TO** Authentication

**MEANS-END FROM** BiometicsAuthentication **TO** Authentication

**MEANS-END FROM** CardkeyAuthentication **TO** Authentication

**MEANS-END FROM** PasswordAuthenication **TO** Authentication

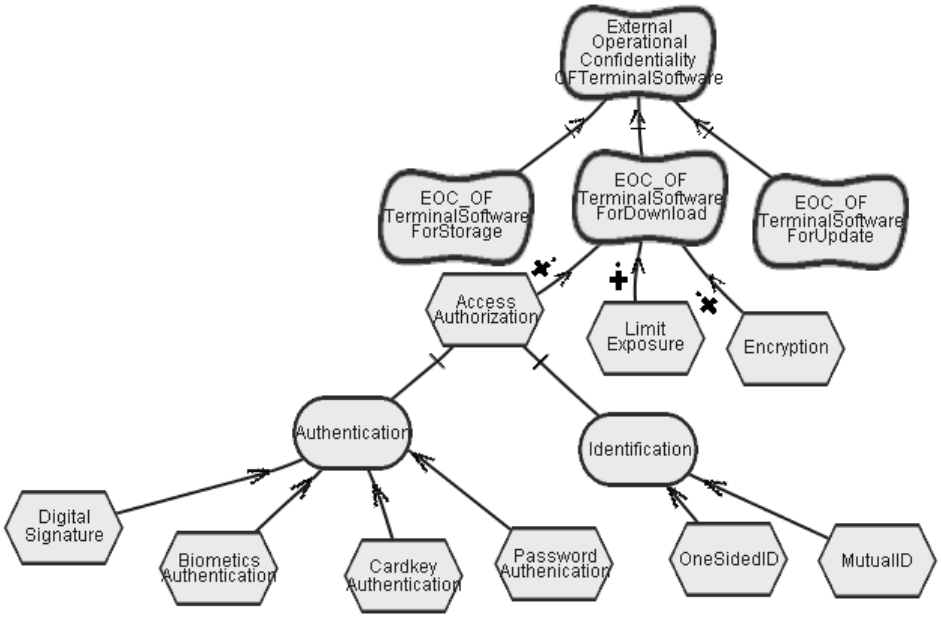
The GOAL Identification can also be provided in two different ways:

**TASK** OneSidedID

**TASK** MutualID

**MEANS-END FROM** OneSidedID **TO** Identification

**MEANS-END FROM** MutualID **TO** Identification




Each one of the alternatives discussed is able to address the confidentiality requirements for updating the terminal software. However, each alternative has a different set of tradeoffs among other quality requirements that might be of importance. For example, using BiometricsAuthentication would provide a high-level of security, however, would be very expensive to provide. On the other hand, using a Cardkey Authentication is less expensive, but still needs some equipment and very user friendly. Finally, using password protection is the least expensive but has not the same user friendliness as the cardkey reader. We would model this kind of reasoning through correlation links to other softgoals.

Finally, in order to demonstrate the Belief construct, let us add a justification for our claim that using Biometrics has very negative impact on the purchase cost of the terminal. We support the very negative impact by the belief that "Biometrics is no regular off-the-shelf technology" and thus still very expensive to provide. Since the Belief is linked to a correlation link, we override the machine-generated identifier and provide a correlation link identifier of our own. This identifier is then used in the BELIEF clause.

**ELEMENT** TerminalSystem

**SOFTGOAL** PurchaseCost **OF** TerminalSystem

**ELEMENT** TerminalSystemUserInterface

**SOFTGOAL** UserFriendliness **OF** TerminalSystemUserInterface

Note that we specify here the correlation link identifier and name it BiometricAuthenticationCorrelationLink. This name is then used when declaring the Belief intentional element.

**CORRELATION** BiometricAuthenticationCorrelationLink **IS**

BiometricsAuthentication

**HAS** Break **CONTRIBUTION-TO** PurchaseCostOFTerminalSystem

**CORRELATION**

CardKeyAuthentication

**HAS** Some- **CONTRIBUTION-TO** PurchaseCostOFTerminalSystem

**CORRELATION**

CardKeyAuthentication

**HAS** Make **CONTRIBUTION-TO** UserFriendlinessOFTerminalSystemUserInterface

**CORRELATION**

PasswordAuthentication

**HAS** Make **CONTRIBUTION-TO** PurchaseCostOFTerminalSystem

**CORRELATION**

PasswordAuthentication

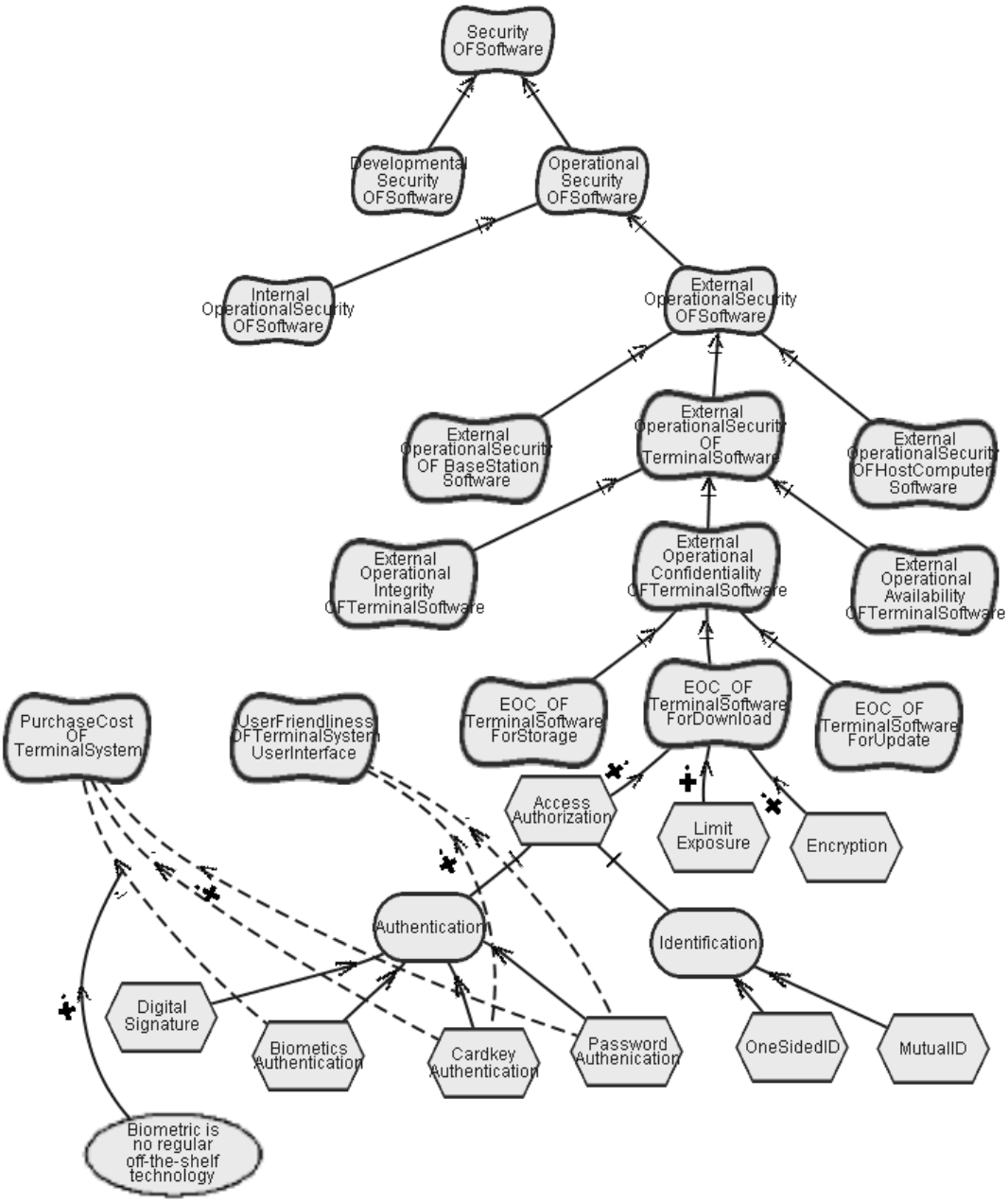
**HAS** Some- **CONTRIBUTION-TO** UserFriendlinessOFTerminalSystemUserInterface

This Belief link is added at the bottom of the following diagram.

The textual description of this link is:

**BELIEF** BiometricAuthenticationCorrelationLink "Biometric is no regular off-the-shelf technology".

All the correlations described provide a basis for evaluating the alternative approaches that may be taken to ensure confidentiality, i.e., unauthorised disclosure of update and related information of the terminal software updates.



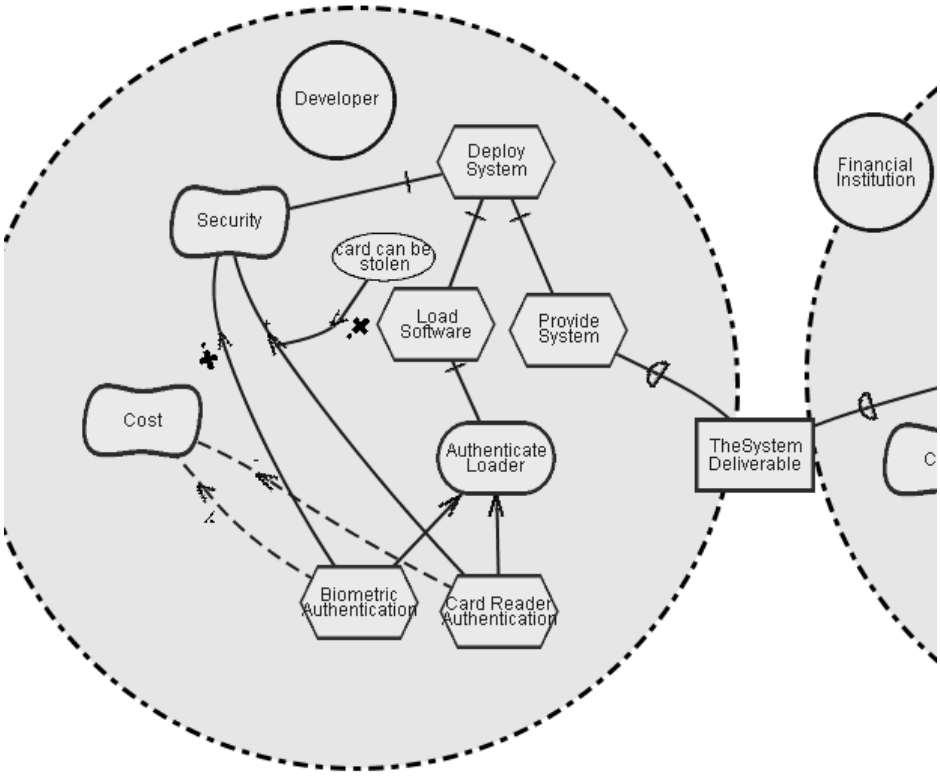

In order to illustrate actor related features of GRL, let us consider the following diagram. We wish to show two Actors participating in the deployment process described in case 1. Both Actors are concerned about security; however, each one may make different choices, based on actor context-dependent considerations.

The first diagram shows the Developer actor (i.e. the people within the development organization), and parts of the process to deploy the software system to the financial institution. During this process, first the developed software is loading onto special purpose hardware (that was supplied by the manufacturer), and then the system deliverables (both software and hardware) is provide to the financial institution.

The diagram shows that there are security concerns related to the deploying of the system. This security requirement is addressed by requiring authentication before loading of the software onto the hardware can commence. The AuthenticateLoader GOAL shows that two alternatives authentication options are considered: use Biometrics or use a Card Reader. As we have seen in the previous example each alterative has different tradeoffs. Biometrics authentication provides very good security. This is denoted by the Make contribution link from BiometricAuthentication Task to the Security Softgoal. It is, however very costly, which is denoted by the Hurt correlation link from the BiomtricAuthentication Task to the Cost Softgoal. The CardRead authentication provides good security, albeit to a lesser extent than Biometric authentication. This is denoted by the Some+ contribution link between CardReaderAuthentication Task and the Security Softgoal. Card Reader technology is, however, less expensive. This is denoted by the Some- correlation link between CardReaderAuthentication Task and the Cost Softgoal. Note that a Belief node is added to explain why Card Reader Authentication is less secure, since cards can be stolen, which introduces a concern not applicable to Biometric Authentication.

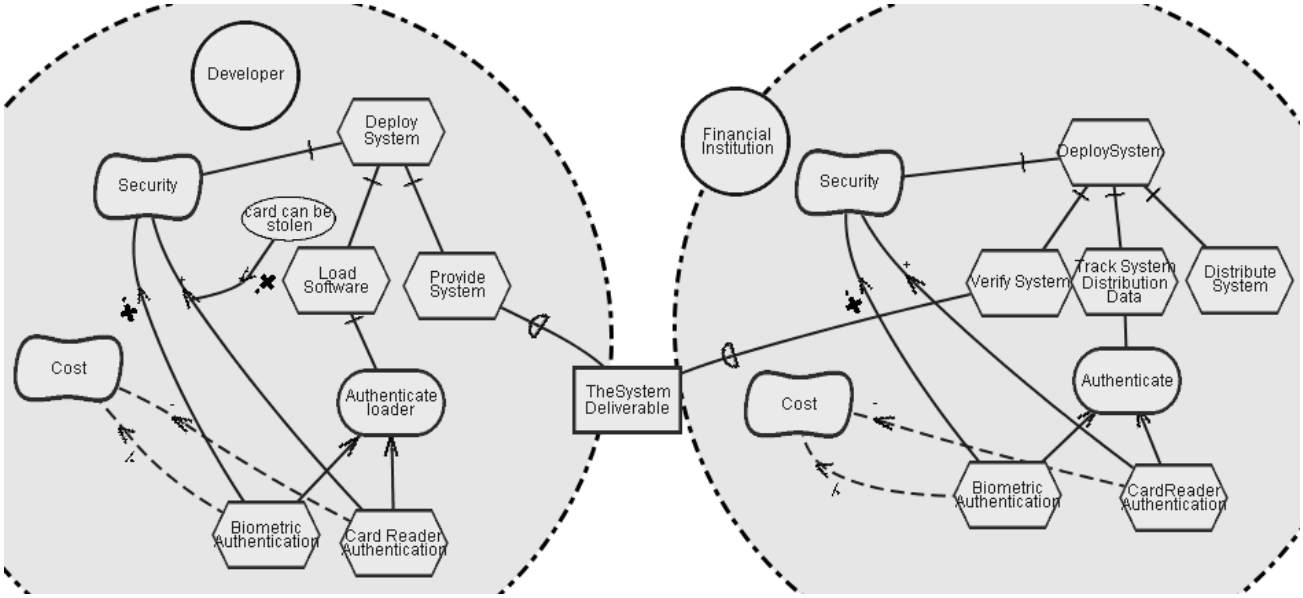
Note that contribution links are used to denote the primary reason for the choice of Tasks, while correlation links denote "side-effects" that relate to these Tasks. Finally, the diagram also adds the Belief "Card can be stolen" to support the fact that CardReaders are only providing "some+" contribution to the Security Softgoal.

This diagram illustrates that reasoning related to Softgoals may be "context dependent" and needs to be considered within the boundaries of an Actor. For the Developer Actor, we can argue that the Development organization is very concerned about Security. It may even be an important factor to being chosen a supplier of the financial institution. They would therefore rather choose Biometric Authentication than Card Readers, although it is the more expensive authentication method.




The next diagram shows the Financial institution Actor now receives the system deliverables and wishes to deploy the system further to its Retailer clients. During its deployment process the financial institution first verifies the system deliverables. This is denoted by the VerifySystem Task. Note that this task is dependent on receiving the system deliverables from the Developers. This dependency is denoted by the TheSystemDeliverable Resource dependency link between the Developer actor and the Financial Institution Actor. The next step in the financial institutions' deployment process, is to data relate to the distribution of the system. This data involves among others handling of the systems software to be distributed. Similarly to the Developers Actor, the Financial Institution is concerned with Security, and wishes to Authenticate users who deal with the tracking system. In this example, the same alternatives for authentication

are considered. However, the financial institution may argue, that since its premises are rigorously secured anyway, Biometrics would be "overkill", while a CardReader authentication system might be enough for securing the access to the software code that is uploaded for further distribution (within the tracking system). Note that if the Security Institution would already have Card Readers installed on site, then using a card reader would have a synergistic effect on the costs involved for authentication. Since such systems are already widely used anyway. This consideration was, however, omitted from the diagram, and would have involved a correlation link between the CardReaderAuthentication Task and the Cost Softgoal of some positive extent.




## B.2 URN-FR tutorial

The URN-FR notation is used for describing *causal relationships* between *responsibilities*, which may potentially be bound to underlying organizational structures of abstract *components*. The resulting visual scenarios are called *Use Case Maps* (UCMs) or simply *maps*. Responsibilities are generic and can represent actions, activities, operations, tasks to perform, and so on. Components are also generic and can represent software entities (objects, processes, databases, servers, functional entities, network entities, etc.) as well as non-software entities (e.g. users, actors, processors). The relationships are said to be causal because they involve concurrency and partial orderings of activities and because they link causes (e.g., preconditions and triggering events) to effects (for example post-conditions and resulting events). In a way, UCMs show related use cases in a map-like diagram.

URN-FR uses behaviour as a concrete, first-class architectural concept. Maps usually emphasize the most relevant, interesting, and critical functionalities of the system, even when little design information is available. With the URN-FR notation, scenarios are expressed above the level of messages exchanged between components; hence they are not necessarily bound to a specific underlying structure. URN-FR provides a path-centric view of system functionalities and improves the level of reusability of scenarios. The notation also enables scenario integration at various levels, architectural reasoning, and description of dynamic behaviour and architectures.

In Stage 1 documents, requirements usually suffer from heavy instabilities, whereas scenarios and potential component topologies are volatile. UCMs fit well in approaches that intend to bridge the gap between requirements and an abstract system design (Stage 2), where a tentative distribution of system behaviour over a structure is being introduced. The generation of Stage 2 documents, which usually contain information flows or Message Sequence Charts, from URN-FR descriptions will hence briefly be addressed. Stage 3 documents describe the various protocols and procedures involved in the support of the MSCs described in the previous stage.

This tutorial illustrates some useful applications of the URN-FR notation to the following steps of systems and standards development processes:

- Description of causal scenarios and architectures (Section B.2.1)
- Architectural reasoning (Sections B.2.1 and B.2.2)
- Refinements with Message Sequence Charts (Section B.2.2)
- Scenario integration (Section B.2.3)

- Description of highly dynamic systems (Section B.2.4)

Several telecommunication services and features will serve as illustrative examples.

## B.2.1 Description of causal scenarios and architectures

### B.2.1.1 Description of a simplified call screening scenario

The URN-FR notation is used to describe causal scenarios when little design information is available. The following example is based on a simplified version of a call-screening feature in a Wireless Intelligent Network system. Figure 20 shows a map where an incoming call is tentatively initiated (IncomingCall, or IC). This causes a screening function to be executed according to the subscriber's policies. In this map, two alternative results are considered. If the call initiator is on the receiver's screening list (condition OnList), then an announcement is played (PlayBlockAnnounce, or PBA) and the call is blocked (CalledBlocked, or CB). Otherwise (condition NotOnList), there is no special treatment (NormalAlerting, or NA) and the call is accepted (CallSetup, or CS). In order for the figures and labels to be more concise, the rest of the section will use abbreviations as labels and will not display the conditions on the OR-fork.

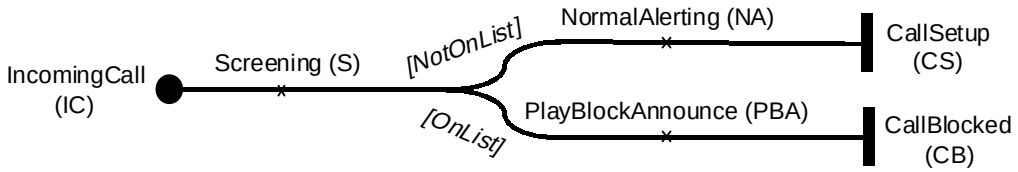


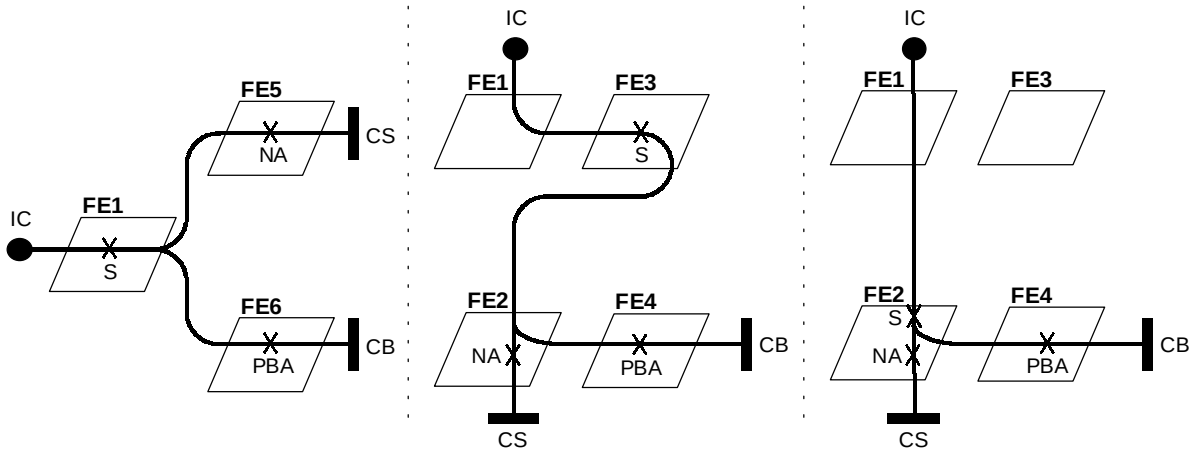

**Figure 20.** URN-FR description of a simplified call screening scenario

A UCM starts with a triggering event or a pre-condition (filled circle labeled IC) and ends with one or more resulting events or post-conditions (bars), in our case CS or CB. A *route* is a scenario that evolves on a path linking an initial cause to a final effect. Intermediate *responsibilities* (S, NA, and PBA) are activated along the way to form routes such as <IC, S, NA, CS>. Think of responsibilities as tasks to be performed. The notation allows for alternative paths (the *OR-Fork* in the figure), concurrent paths, exception paths, timers, stubs with plug-ins, and synchronous or asynchronous interactions between paths.

Such maps have proved to be very useful in Stage 1 descriptions of service functionalities. Their principal emphasis is on causality and responsibilities, without any reference to messages, system components, or component states. Yet, they represent useful and powerful tools for the support of the thinking process and the evaluation of functional alternatives. This causal dependence between responsibilities should be documented as early as possible in the design process, before this information gets lost among the details of the behaviour of individual components. This is especially true of concurrent, communicating, and distributed systems.

### B.2.1.2 Evaluation of architectural alternatives for functional entities

UCM paths can be bound to various structures of components, which leads to a visual integration of scenarios and architecture in a single view. The URN-FR notation supports the reuse of scenarios when the underlying structure is modified or refined, even across architectural domains. For instance, Figure 21(a) and Figure 21(b) illustrate two sample structures of *functional entities* (FEs), which are the components of IN's distributed functional plane. In this example, FEs are shown as processes, i.e. with parallelograms. The example scenario (Figure 20) is bound differently to each collection of FEs. The same scenario can also be bound to the same structure, but in a different way, as shown in Figure 21(c).




(**a**) First structure (**b**) Second structure (**c**) Different mapping of UCM

**Figure 21.** A causal scenario path bound to different structures of FEs

In Stage 2 descriptions, different potential structures could undergo some evaluation, hence enabling architectural reasoning. Scenarios described in terms of wired components, such as Message Sequence Charts or as sequence diagrams in the Unified Modelling Language (UML), would need to be rebuilt as soon as there is a change in the underlying structure, because the functionalities are tightly bound to how the structure looks like. UCMs scenarios are more reusable and require simpler modifications, consisting only of a new binding between the responsibilities and the components.

### B.2.2 Refinements with Message Sequence Charts

When extracting MSC-like scenarios directly from informal requirements, as it is often done in conventional system development, many design decisions become buried in the details of the scenarios. For instance, Wireless Intelligent Network MSCs are often described in terms of network entities, which contain various functional entities. Design decisions such as the allocation of UCM responsibilities to functional entities (the logical components in the distributed functional model) and the allocation of functional entities to network entities (the physical entities in the network reference model) are lost. Assuming that such a standard does not impose a specific mapping of functional entities to network entities, different vendors who build network entities may use different mappings. Designers must reverse-engineer information and scenarios that would be explicit in a UCM view where responsibilities and other constructs are bound to functional entities. This delays the design and implementation phases and leads to multiple interoperability problems. Such problems are unfortunately common in standards.

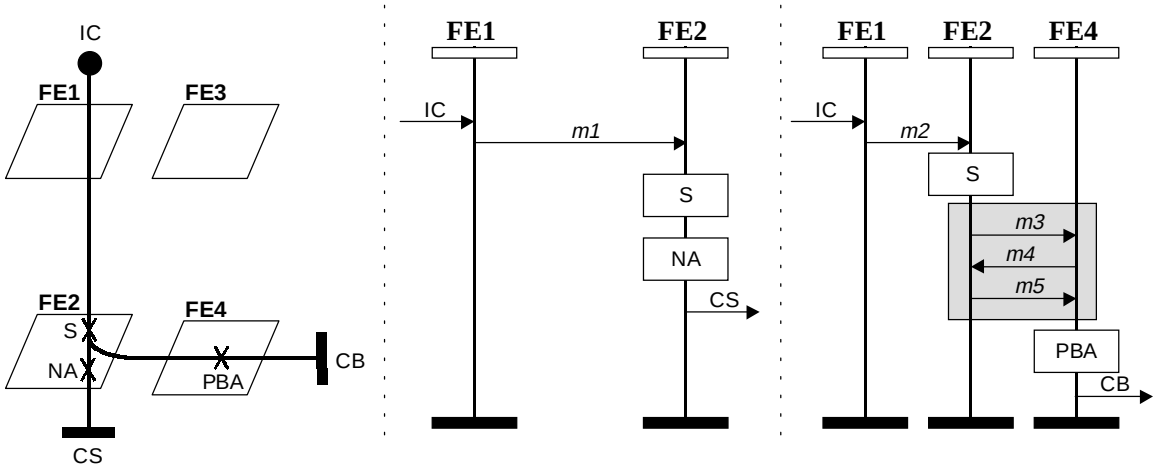
By using a UCM view, many issues related to messages, protocols, communication constraints, and structural evolutions (e.g. from one version of the structure to the next) can be abstracted, and the focus can be put on intended functionalities and on reusable causal scenarios in their structural context. This section aims to illustrate the kind of design decisions that one needs to take and document when generating MSCs from URN-FR descriptions.

#### B.2.2.1 MSC refinements of bound maps

Bound UCMs can serve as a basis for the generation of correct MSCs in Stage 2 documents, provided that additional information related to communication constraints, protocols, and parameters are defined to refine inter-component causal relationships into message exchanges. Many MSCs could be valid according to a UCM, as long as the intended causal relationships between the responsibilities are satisfied. The following example uses Figure 21(c), duplicated in Figure 22(a), as a starting point.

Figure 22(b) is a potential MSC extracted from the route <IC, S, NA, CS>, where the exchange of messages is minimal. Responsibilities are performed by the entities to which they are bound, so S and NA are actions performed by FE2. In this example, input and output events, captured by the start point IC and the end point CS, are interpreted as messages from and to the environment. The causal path linking FE1 to FE2 is refined by an appropriate exchange of messages between these two entities (e.g. the simple message *m1*). Different protocols could be used at this point, and the flow of data would need to be specified as well.

Figure 22(c) is a potential MSC extracted from the route <IC, S, PBA, CB>. As implied by the multiple messages between FE2 to FE4, complex protocols or negotiation mechanisms could be involved between two given entities.

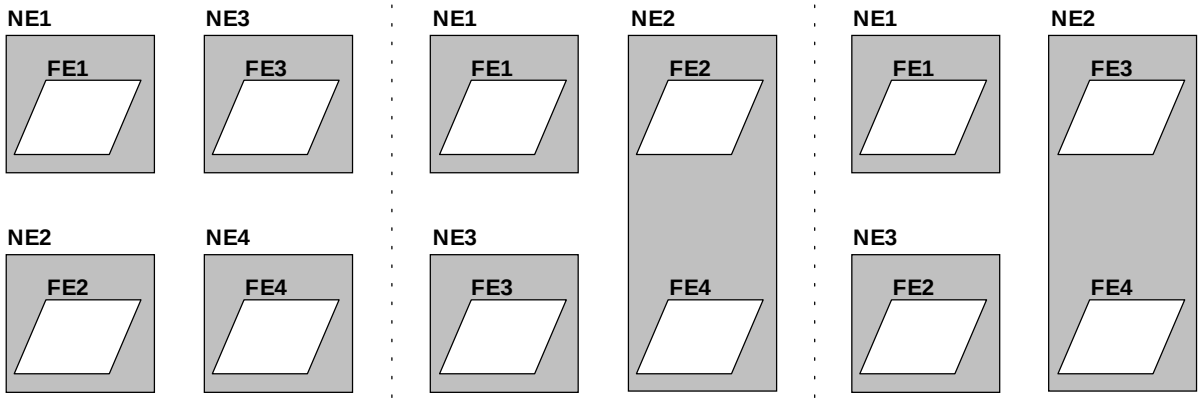



(**a**) UCM to FEs (**b**) A MSC for <IC,S,NA,CS> (**c**) A MSC for <IC,S,PBA,CB>

**Figure 22.** Generation of potential MSCs from scenario paths bound to a structure of FEs

### B.2.2.2 Structural alternatives for functional entities and network entities

Moving from Stage 2 documents to Stage 3 documents requires for the functional entities to be bound to network entities (NEs), the components of IN's physical plane. Again, different allocations are possible, and design decisions have to be made and documented. Two collections of NEs lead to different mappings in Figure 23(a) and Figure 23(b). In Figure 23(c), the structure remains the same as in Figure 23(b), but FE2 and FE3 are allocated differently.



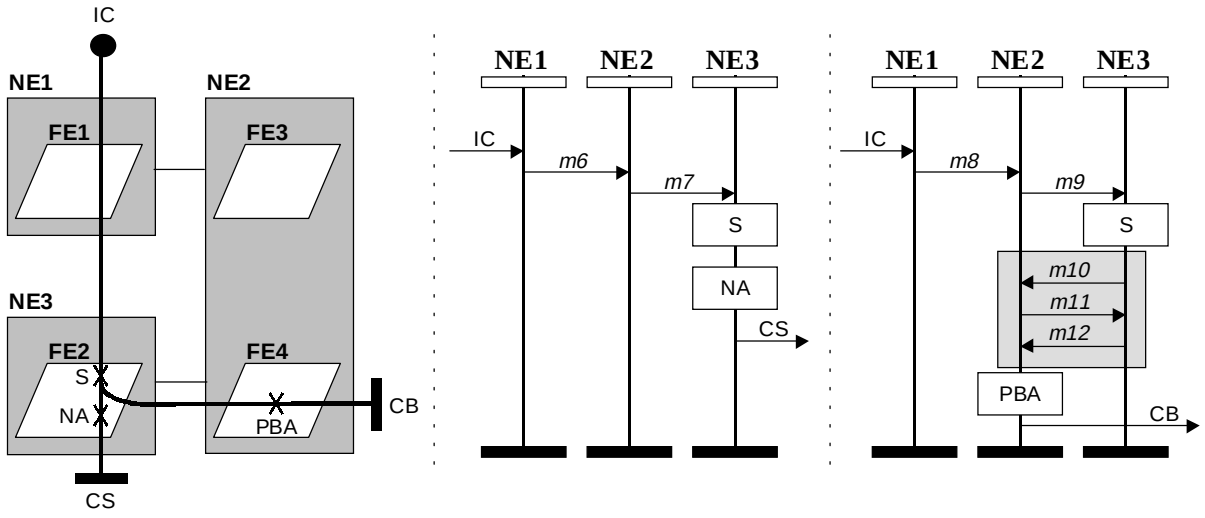

(**a**) First structure (**b**) Second structure (**c**) Different mapping of FEs

**Figure 23.** Different bindings of functional entities to network entities

### B.2.2.3 MSC refinements of bound maps with constrained communication mechanisms

The bound map in Figure 24(a) results from the allocation of responsibilities to FEs described in Figure 21(c) and from the allocation of FEs to NEs described in Figure 23(c). Components may have restricted access to each other. This type of constraint is often defined by communication channels or interfaces linking the components. In this example, NE2 is allowed to communicate with both NE1 and NE3, but NE1 and NE3 cannot communicate directly. This is shown by the links between the various components (not part of the URN notation). Such constraints are often described in Stage 3 documents, hence paving the way towards protocols and procedures for the final architecture.

At this level, a new collection of MSCs can be generated, this time in terms of network entities. Figure 24(b) is a potential MSC extracted from the route <IC, S, NA, CS>, where the exchange of messages is minimal. Notice how NE2 is used to refine the causal relationship between IC and S: NE1 is not allowed to send messages directly to NE3, but messages can be forwarded through NE2. Similarly, Figure 24(c) is a potential MSC extracted from the route <IC, S, PBA, CB>, this time with a complex negotiation mechanism between NE2 and NE3, which may result from the negotiation between FEs as suggested by Figure 22(c). Again, those MSCs are just potential candidates. Many other MSCs could refine the same map according to different choices in the communication constraints and protocols.




(**a**) UCM to FEs to NEs (**b**) A MSC for <IC,S,NA,CS> (**c**) A MSC for <IC,S,PBA,CB>

**Figure 24.** Generation of MSCs from scenario paths bound to a constrained structure of NEs

The types of decision discussed in this section are only relevant in a late stage of the development process. Yet, many standards and system descriptions include MSCs based on network entities (or the equivalent) in Stage 2, and sometimes even in preliminary Stage 1 documents, hence skipping many decision steps. This hurts the development process by not separating concerns, by narrowing the number of potential implementations too soon, and by not documenting any of the requirements and design decisions needed to enable adaptation and evolution.

### B.2.3 Integration of scenarios

Scenarios are often described in isolation. However, at some point in time, a view where all scenarios are integrated together becomes important for analysis, for consistency checking, for the detection of interactions, and for the construction of more detailed models. This section contains a second example (unrelated to the call screening feature discussed so far) where three features, described as individual maps based on a simplified call initiation, are integrated into one map with sub-maps (plug-ins).

#### B.2.3.1 Simplified call initiation scenario

The following call initiation scenario is bound to a structure with two types of components, User and Agent, which can both assume one of two roles: originating (O) or terminating (T). The originating user initiates a call through the start point req (Figure 25). The terminating agent verifies whether the terminating user is busy or idle (vrfy). When idle, the terminating user status is updated in its agent (upd) and a resulting ringing event occurs at the terminating user side (ring). Simultaneously, this results in an event at the originating user side (sig) signaling a prepared ringback reply (prb). When busy, a busy reply is prepared (pb) and a corresponding signaling event results at the originating user side.

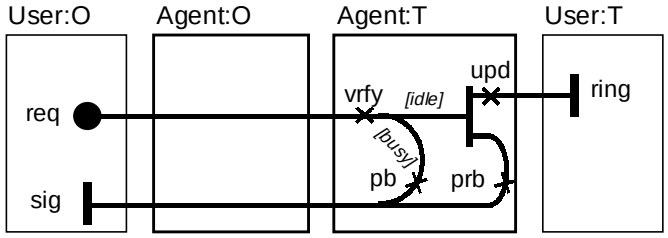



**Figure 25.** Simplified call initiation UCM

This UCM already integrates multiple sequential scenarios through path elements like OR-forks and AND-forks.

#### B.2.3.2 Three features described in isolation

Often, telecommunications features are described in terms of an underlying basic call. The following three features extend the simplified basic call UCM to include new functionalities or prevent existing ones.

The first feature is the originating call screening (OCS), described in Figure 26. The originating agent possesses a screening list object (OCSlist) that is checked (chk) to determine whether the call should be allowed or denied. When allowed, the call initiation scenario continues. When denied, a denied reply is prepared (pd) and a corresponding signaling event results at the originating user side.

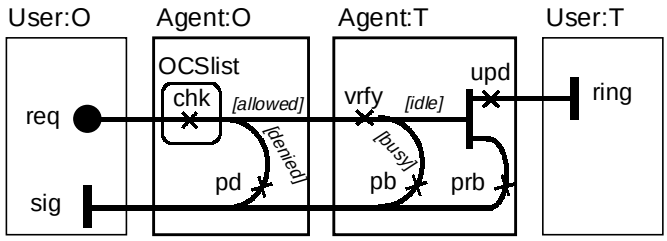



**Figure 26.** Originating call screening (OCS) feature UCM

The UCM in Figure 27 describes the TEENLINE feature, located in the originating agent. This feature checks the current time (chkTime) to determine whether the call is being initiated at a specific time of the day. When in the predefined range, TEENLINE requires a valid personal identification number (PIN) to be provided in a timely fashion for the call initiation to continue. If an invalid PIN is provided, or if a time-out occurs, then a denied reply is prepared (pd).

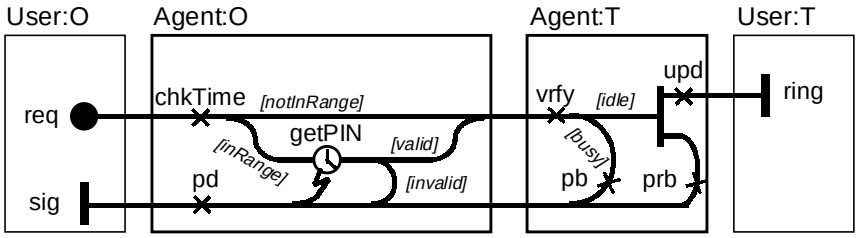



**Figure 27.** TEENLINE feature UCM

Figure 28 presents the third feature, Call number delivery (CND), located at the terminating side. This feature displays (disp) the name of the calling party, i.e. the originating user.

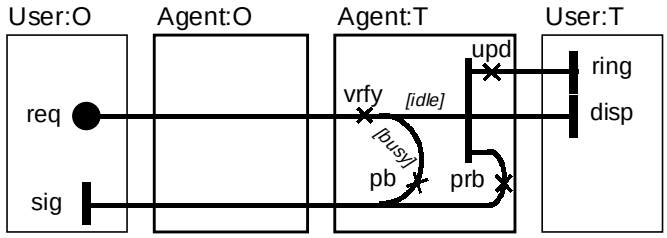



**Figure 28.** Call number delivery (CND) feature UCM

### B.2.3.3 Integrated scenarios

UCMs can help structuring and integrating scenarios in various ways, e.g. sequentially, as alternatives (with OR-forks/joins) or concurrently (with AND-forks/joins). However, one of the most interesting constructs for scenario integration is certainly the stub. While *static stubs* contain only one sub-map (plug-in), *dynamic stubs* may contain multiple sub-maps whose selection can be determined at run-time according to a *selection policy*. Such a policy can make use of preconditions, assertions, run-time information, composition operators, etc. in order to select the plug-in(s) to use. Selection policies are described with a (formal or informal) language suitable for the context where they are used. The plug-in maps are sub-maps that describe locally how a feature modifies the basic behaviour. Multiple levels of stubs and plug-ins can be used.

A potential root map that integrates the three features seen so far is presented in Figure 29. Stub SO contains the originating features whereas stub ST contains a single plug-in (illustrating scenario decomposition), which in turn will possess a stub (SD) for the terminating features. Each of these stubs includes plug-ins that represent how the call initiation reacts in the presence or absence of features.

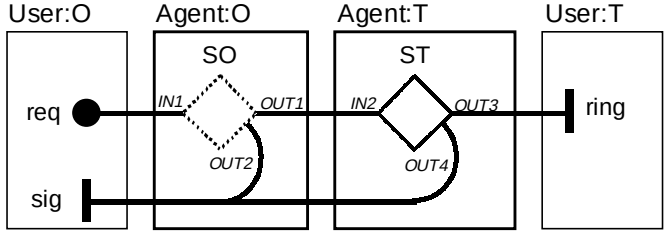
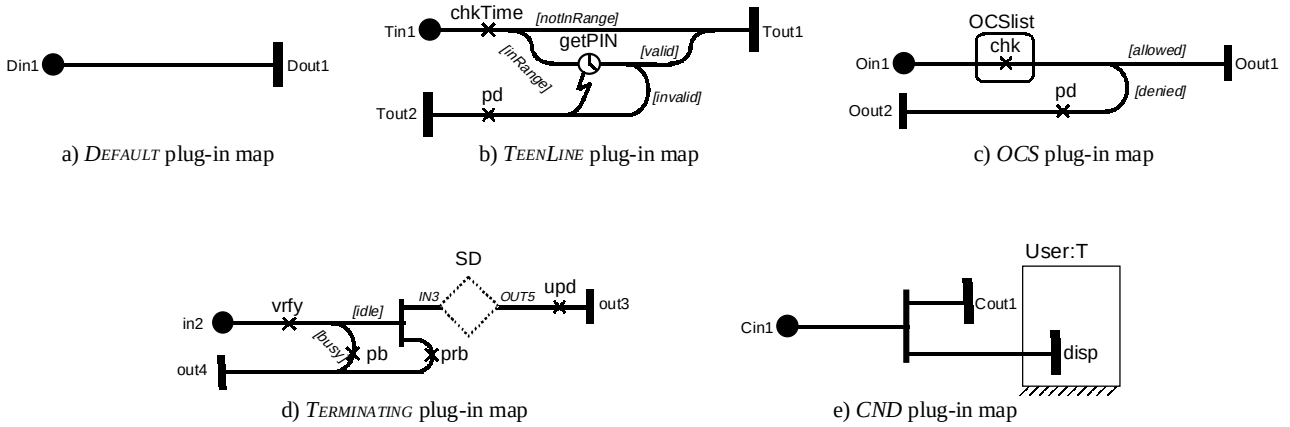



**Figure 29.** ROOT map for integrated call initiation

Five plug-ins are presented in Figure 30. These sub-maps are connected directly or indirectly to the ROOT map of Figure 29 in order to support the features and the basic call initiation. By default, these plug-ins act locally in the component where the parent stub is located. If a component outside the containing component is referenced by the plug-in (e.g. User:T in Figure 30(e)), then this component needs to be declared as *anchored*. Such components have a shadow at the bottom to indicate visually that they are already defined elsewhere.




a) *DEFAULT* plug-in map b) *TEENLINE* plug-in map c) *OCS* plug-in map

d) *TERMINATING* plug-in map e) *CND* plug-in map

**Figure 30.** Plug-ins for the various features

The binding relationships, which are used to connect plug-ins to stubs, are enumerated in Table 6. Through these relationships, it is possible to reconstruct the initial UCMs:

- Simplified call initiation (Figure 25): DEFAULT in SO, TERMINATING in ST, DEFAULT in SD.
- OCS feature (Figure 26): OCS in SO, TERMINATING in ST, DEFAULT in SD.
- TEENLINE feature (Figure 27): TEENLINE in SO, TERMINATING in ST, DEFAULT in SD.
- CND feature (Figure 28): DEFAULT in SO, TERMINATING in ST, CND in SD.

**Table 6** Binding relationships for integrated view

| Parent map | Stub | Plug-in map | Figure 30 | Binding relationship |
|---|---|---|---|---|
| ROOT | SO | DEFAULT | (a) | {<IN1, Din1>, <OUT1, Dout1>} |
| | | TEENLINE | (b) | {<IN1, Tin1>, <OUT1, Tout1>, <OUT2, Tout2>} |
| | | OCS | (c) | {<IN1, Oin1>, <OUT1, Oout1>, <OUT2, Oout2>} |
| | ST | TERMINATING | (d) | {<IN2, in2>, <OUT3, out3>, <OUT4, out4>} |
| TERMINATING | SD | DEFAULT | (a) | {<IN3, Din1>, <OUT5, Dout1>} |
| | | CND | (e) | {<IN3, Cin1>, <OUT5, Cout1>} |

Adding features to such UCM collections is often achieved by creating new plug-ins for the existing stubs, or by adding new stubs containing either new plug-ins or instances of existing plug-ins. In all cases, the selection policies need to be updated appropriately.

#### B.2.3.4 Feature interactions and selection policies

Dynamic stubs contain selection policies used to determine which plug-in(s) should be selected at run-time. For stub SD, a potential policy could be as simple as this condition:

**IF** subscribed(User:T, CND)

**THEN** select CND

**ELSE** select DEFAULT

**ENDIF**

However, things are slightly more complex for stub SD, which contains three plug-ins. Both OCS and TEENLINE can easily have priority over DEFAULT, but there exists an interaction between these two features that requires a resolution. Stubs and selection policies tend to localize the places on scenarios where undesirable feature interactions can occur, hence simplifying the analysis and the resolution of these undesirable interactions. Among other things, they can be used to specify priorities of some features over others. For instance, TEENLINE could be given precedence over OCS in stub

SO. One feature could even prevent another feature from being executed. Many spurious interactions between features are thus avoided by structuring and integrating the scenarios in the proper context.

In our particular example, one potential policy could be:

**IF** subscribed(User:O, OCS)

**THEN** **IF** subscribed(User:O, TEENLINE)

**THEN** select TEENLINE ; select OCS

**ELSE** select OCS

**ENDIF**

**ELSE** **IF** subscribed(User:O, TEENLINE)

**THEN** select TEENLINE

**ELSE** select DEFAULT

**ENDIF**

**ELSE** select DEFAULT

**ENDIF**

Although it works in this case, this type of policy does not scale well or large numbers of plug-ins. Mechanisms based on assertions and concurrent negotiation between plug-ins have the potential of coping with this issue.

### B.2.4 Description of highly dynamic systems

Highly dynamic systems are usually difficult to represent using a static notation. Section B.2.3 already presented how dynamically "pluggable" behaviour can be described using dynamic stubs. However, describing self-modifying architecture and mobile behaviour remains a major challenge. The following two examples present how URN-FR can be used to address this issue.

#### B.2.4.1 Description of dynamic architectures

This example makes use of slots, pools, and dynamic responsibilities to demonstrate the expressiveness of these concepts in a dynamic architecture context. A hypothetical telephony system is composed of a telephone switch, containing multiple devices, and of a provisioning site responsible for the creation, storage, and retrieval of device handlers (Figure 31). The switch contains devices, which perform telephony operations, and a driver process, which requests and installs required device handlers. The provisioning site contains a server process that decodes requests for new handlers, and a supplier process where new device handlers are created. These handlers are stored for future use in a repository, represented in the map by a pool of objects.

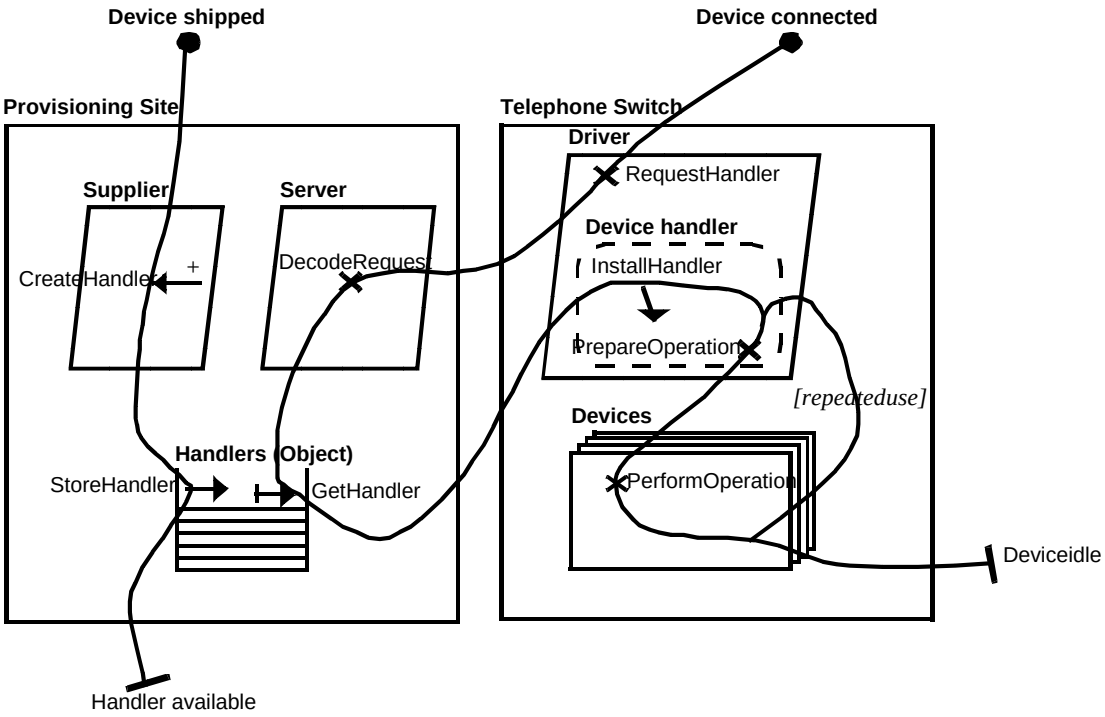



**Figure 31.** Dynamic device handler map

The scenario path on the left can be interpreted intuitively as follow. Upon the shipment of a new device, Supplier creates a new handler object and then stores it in the Handlers pool. At this point, this new handler is available to the switch. The scenario path on the right indicates that once the real device is connected to the switch, the Driver can request the appropriate handler from the provisioning site (this is in a sense similar to a plug-and-play device). The Server decodes the request and gets the handler object from the pool. This object is then installed in the Device handler *slot*, a placeholder that cannot do anything until a component gets installed. At this point, the installed device handler can prepare whatever operation is requested from a number of devices, which then get to perform the operation. The use of the handler can be repeated until no operations are required, and then the device(s) become idle.

### B.2.4.2 Description of mobile behaviour

Dynamic responsibilities and pools can also be used to describe mobile behaviour that can be created, stored, retrieved, and installed in dynamic stubs. Figure 32 illustrates this concept through an example that involves a telephone switch and a service creation environment. Similar to Figure 31, the path on the left enables the creation and storage of new services (plug-ins) that are made available to the switch. The latter can dynamically request such service from the service creation environment, plug it in a dynamic stub, and initiate the execution of this service (in PerformService).

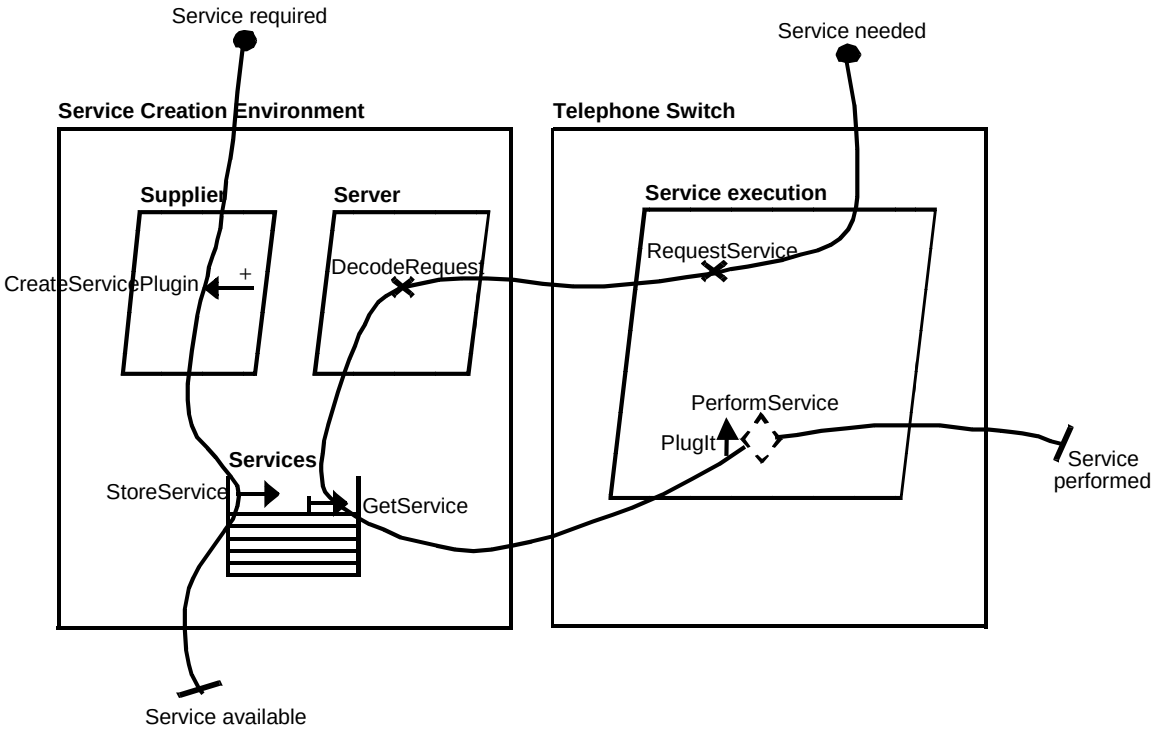



**Figure 32.** Service creation and execution map

The URN-FR notation thus proposes powerful constructs (dynamic stubs, dynamic responsibilities, pools, and slots) to capture and describe, in a concise way, complex scenarios that involve highly dynamic behaviour and architectures. This enables the representation of systems that involve agents and mobile code, two important features of the upcoming generation of distributed systems in general and telecommunications systems in particular.

# Annex C
# Relationships between URN and other notations

## C.1 Translation of URN-FR into other notations

### C.1.1 Message Sequence Charts (MSC)

This section contains a description of the experience gained translating the URN-FR into the MSC notation.

### C.1.2 Layered Network Queuing (LQN) Notation

This section contains a description of the experience gained translating the URN-FR into the LQN notation.

### C.1.3 Specification and Description Language (SDL)

This section contains a description of the experience gained translating the URN-FR into SDL.

### C.1.4 Language Of Temporal Ordering Specification (LOTOS)

This section contains a description of the experience gained translating the URN-FR into LOTOS.

### C.1.5 Unified Modelling Language (UML)

This section contains a description of the experience gained translating the URN-FR into UML.

### C.1.6 Tree and Tabular Combined Notation (TTCN)

This section contains a description of the experience gained translating the URN-FR into TTCN.

TO BE DONE

# Appendix I
# Requirements engineering activities

This annex discusses the major set of Activities related to Requirements Engineering and how these relate to a goal-oriented approach.

Requirement analysis (and more broadly requirements engineering) covers multiple intertwined activities to arrive at requirements specification of the intended software system. We can suggest the following major activities involved:

- Domain or early requirements analysis
- Eliciting requirements
- Modeling and analyzing requirements
- Documenting and communicating requirements among stakeholders over the software system life-cycle
- Agreeing on and validating requirements
- Specification
- Specification analysis
- Verifying, Managing and Evolving requirements

Each of these activities has distinct concerns and may be helped by a goal-oriented and process-oriented URN.

**Domain or early requirements analysis:** During this phase the existing environment in which the software system should be built is studied, and the relevant stakeholders, who are affected by the intended system are identified. This may include the intended users, the clients who commissioned the system, 3rd parties, and stakeholders within the development organisation, international standard bodies, and the like. At this early stage the stakeholders' interests and how they might be addressed, or compromised, by various system-and-environment alternatives is explored. Each alternative may explore different boundaries between the software-to-be and its environment.

Providing for goals (and to a certain extent, for a representation of stakeholders) within the URN allows capturing the high-level objectives of pertinent stakeholders. Linking those goals to elements of the requirements specification, design and implementation, aids in better understanding and managing how changes in these high-level objectives affect the system during development, maintenance and evolution. Capturing and linking goals in such a way, also aids in dealing with understanding how software systems either facilitate or hinder co-operation among organisations that wish to create alliances in order to pursue co-operative objectives.

**Eliciting Requirements:** During this phase alternative models for the target system are further elaborated and explored such that stakeholders' objectives are met. Particular requirements and assumptions on the various organisational actors who would interact with the intended software system, the system, and to a certain extend pertinent high-level components of the system are established. Often the users of software systems are not able to articulate their requirements. Describing the tasks, scenarios or use cases for the current and/or the intended system can help users in making their requirements explicit.

Goals and the ability to refine them towards potential alternative system specifications may provide guidance when eliciting requirements. Knowledge-based approaches that capture know-how of achieving goals may be invoked during the elicitation process, to suggest further goal refinements, certain functional, structural and organisational requirements for achieving goals. The synergy between scenarios and goals is discussed in the literature. Linking goals to scenarios facilitates checking if all goals have been met, thus establishing completeness of the requirements specification. The ability to ask "why" for scenarios may provide opportunities to identify new goals, while "how else" questions may yield alternative scenarios for achieving goals.

**Modelling and analysis:** This is an activity that is found during all phases of requirements engineering and appears to be a core process in requirements engineering. The existing system needs to be modelled in one way or another, while the hypothetical alternative systems need to be modelled as well. Models serve as the basic common interface for the various requirements engineering activities. Models also provide the basis for documentation and evolution.

The ability to provide goals as an explicit modelling construct together with the ability to refine goals, and link them to various requirements, design and implementation related modelling elements, facilitates making clear how these elements produced in each phase relate to elements of previous phases. In addition, linking elements may provide an anchor point for reasoning about design decisions, justifying or refuting them during the development process. Goals and process support thus provide good groundwork for dealing with requirements evolution and change over the life cycle of the system, i.e., requirements management.

**Documenting and Communicating requirements:** This activity deals with capturing the various decisions made during the RE process, together with their underlying rationale and assumption, and with effectively communicating requirements among stakeholders. Part of the documentation effort that becomes increasingly recognised as crucial is requirement management, which is the ability to not only write requirements but also to do so in a form that is readable and traceable over the life cycle.

Similarly, to the modelling and analysis support, the ability to provide for goals and linking them to requirements, design and implementation, provides traceability links from the source of requirements (i.e. stakeholders' goals) to the requirements specification, design and implementation. Goals would appear in all phases and would be related through refinements links, from high-level organisational goal until low-level design goals. This would also allow for managing change during the development life cycle. Documenting NFRs within a (semi) formal framework also improves the ability to communicate pertinent NFRs among the relevant stakeholders, by clustering them together systematically in hierarchical manners, rather than having them spread out informally within text based documentation.

**Achieving agreement on and validation of requirements:** Since the source for requirements of a software system are the various stakeholders affected and/or involved, disagreement among stakeholders may lead to differences in expectations of what the intended software system should provide. This problem is compounded when stakeholder have divergent goals that they wish to have achieved by the software system. Explicitly describing the stakeholders' objectives and how they relate to the system's requirements specification is a necessary precondition for detecting, negotiating and resolving conflicts among stakeholders. Another aspect of "agreement", besides dealing with conflicts among stakeholders, is the validation of software system requirements. This involves the agreement of stakeholders that the documented requirements are in fact meeting their stated objectives.

Capture of goals with the URN allows early detection of apparent conflicting software system requirements and may, as a result, help to initiate negotiations for arriving at compromises and agreement. Conflicts among organisational and system related goals may surface during the detailed requirements specification, but also during the design and implementation phase, when particular choices for achieving certain goals exclude the ability for achieving others. Life-cycle support within the URN would then enable identifying, and dealing with such conflicts. Negotiation techniques may focus on trying to identify the most important goals of stakeholders, and ensure that they are met, such that the best trade-offs among alternative receives agreement from all parties involved. Goals may aid in determining how and how well the objectives of stakeholders were in fact met by the requirements specification.

**Specification and specification analysis:** These are activities where requirements and assumptions are formulated in precise ways, and checked for deficiencies (such as inadequacies, incompleteness, or inconsistencies) and for feasibility in terms of resources required, development costs etc.

Goals in general and NFRs-related goals in particular may aid here in selecting among alternative specification elements, and in justifying such specification decisions. Analysis of specification may then be placed in context of organisational goals that are (or are not) achieved throughout the detailed specification. Furthermore, prioritising goals may allow choosing particular areas within the system, where precise and formal specification is more desirable that other, often less crucial areas, where a semi-formal or perhaps informal specification is sufficient. Often security or performance NFRs play a role when formal methods are needed to prove such properties for the system.

**Evolution**: The requirements are modified to accommodate corrections, environmental changes, or new objectives. During evolution, both functional behaviour and architecture elements need to accommodate changes in the requirements specification, which often originate from changed organisational objectives**.**

As already elaborated earlier, having the ability to describe rationales for the existence of elements within the specification, design and implementation, supports tracing how changes in the objectives of the organisation affect the rest of the software development. Similarly, how changes in design and implementation artefacts, may affect organisational objectives and their corresponding stakeholders.

A URN should permit the expression of different degrees of formality to reflect the shift in understanding as the users apply refinements to the model during the course of analysis and specification. For example, during early requirements emphasis may be given to relationship among functional and non-functional elements, and stakeholders and their objectives rather than precise definitions of those elements, while during later stages of requirements, more formality would be necessary for providing more precise requirements descriptions.

# Appendix II
# Tool issues

## II.1 URN-NFR

A prototype tool for the URN-NFR notation exists. It is called NFR Framework. This section captures experience gained by the developers of NFR Framework with respect to the user interface and behaviour of the tool.

TO BE DONE.

## II.2 URN-FR

A prototype tool for the URN-FR notation exists. It is called the UCM Navigator. UCM stands for Use Case Map. This section captures experience gained by the developers of the UCM Navigator with respect to the user interface and behaviour of the tool (http://www.UseCaseMaps.org/tools/ucmnav/).

TO BE DONE.

# Appendix III
# External references

This appendix contains references to books, journal papers, and conference papers dealing with requirements engineering and URN.

## III.1 URN-NFR references

This section contains references to books, journal papers and conferences papers dealing with requirements engineering and non-functional requirements.


- Chung, L. and Nixon, B. (1995) "Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach,". In: *Proc, IEEE 17th International Conference on Software Engineering (ICRE'95)*, Seattle, USA, pp. 25-37.
- Chung, L., Nixon, B. and Yu, E. (1997) "Dealing with Change: An Approach Using Non-Functional Requirements". In: *Requirement Engineering*, Springer-Verlag, vol. 1, no. 4, pp. 238-260.
- Chung, L., Gross, D., and Yu, E. (1999) "Architectural Design to Meet Stakeholder Requirements". In: P. Donohue (ed.) *Software Architecture*, Kluwer Academic Publishers. pp. 545-564. Also in: *First Working IFIP Conference on Software Architecture (WICSA1)*, February 1999, San Antonio, Texas, USA.
- Chung, L., Nion, B.A., Yu, E., and Mylopoulos, J. (2000) *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers. ISBN 0-7923-8666-3.
- Gotel, O. and Finkelstein, A. (1994) "An analysis of the requirements traceability problem". In: *First Int. Conference on Requirements Engineering (ICRE'94)*, Colorado Springs, USA, 94-101.
- Mylopoulos, J., Chung, L. and Nixon, B. (1992) "Representing and Using Non-Functional Requirements: A Process-Oriented Approach". In: *IEEE Transactions on Software Engineering, Special Issue on Knowledge Representation and Reasoning in Software Development*, 18(6), June 1992, pp. 483-497.
- Mylopoulos, J., Chung, L. and Yu, E. (1999) "From Object-Oriented to Goal-Oriented Requirements Analysis". In: *Communications of the ACM*, 42(1): 31-37, January 1999.
- Nuseibeh, B. and Easterbrook, S. (2000) "Requirements Engineering: A Roadmap". In: Finkelstein, A. (ed.) *The Future of Software Engineering*. Special track of the *2nd Int. Conference on Software Engineering (ICSE'2000)*, ACM Press.
- van Lamsweerde, A. (2000) "Requirements Engineering in the Year 00: A Research Perspective". In: *Proc. 22nd Int. Conference on Software Engineering (ICSE'2000)*. Limerick, June 2000, ACM press.
- Yu, E. (1997) "Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering". In: *Proceedings of the 3rd IEEE Int. Symposium on Requirements Engineering (RE'97)*, Washington D.C., USA, pp. 226-235. ftp://ftp.cs.toronto.edu/pub/eric/RE97.pdf


## III.2 URN-FR references

This section contains references to books, journal papers and conferences papers dealing with Use Case Maps (UCMs) and topics related to UCMs. The URN-FR language is based on the UCM notation.


- Amyot, D., Andrade, R., Logrippo, L., Sincennes, J., and Yi, Z. (1999) “Formal Methods for Mobility Standards”. In: *IEEE 1999 Emerging Technology Symposium on Wireless Communications & Systems*, Richardson, Texas, USA, April 1999. http://www.UseCaseMaps.org/pub/ets99.pdf
- Amyot, D. and Andrade, R. (1999) “Description of Wireless Intelligent Network Services with Use Case Maps”. In: *SBRC’99, 17º Simpósio Brasileiro de Redes de Computadores*, Salvador, Brazil, May 1999, 418-433. http://www.UseCaseMaps.org/pub/sbrc99.pdf
- Amyot, D., Buhr, R.J.A., Gray, T., and Logrippo, L. (1999) “Use Case Maps for the Capture and Validation of Distributed Systems Requirements”. In: *RE'99, Fourth IEEE International Symposium on Requirements Engineering*, Limerick, Ireland, June 1999, 44-53. http://www.UseCaseMaps.org/pub/re99.pdf
- Amyot, D. and Logrippo, L. (2000) “Use Case Maps and LOTOS for the Prototyping and Validation of a Mobile Group Call System”. In: *Computer Communication*, 23(12), 1135-1157. http://www.UseCaseMaps.org/pub/cc99.pdf

- Amyot, D., Charfi, L., Gorse, N., Gray, T., Logrippo, L., Sincennes, J., Stepien, B., and Ware, T. (2000) "Feature description and feature interaction analysis with Use Case Maps and LOTOS". In: *Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00)*, Glasgow, Scotland, UK, May 2000. http://www.UseCaseMaps.org/pub/fiw00lotos.pdf
- Amyot, D. and Mussbacher, G. (2000) "On the Extension of UML with Use Case Maps Concepts". In: *<<UML>>2000, 3rd International Conference on the Unified Modeling Language*, York, UK, October 2000. http://www.UseCaseMaps.org/pub/uml2000.pdf
- Amyot, D. (2000) "Use Case Maps as a Feature Description Language". In: S. Gilmore and M. Ryan (Eds), *Language Constructs for Designing Features*. Springer-Verlag. http://www.UseCaseMaps.org/pub/fireworks2000.pdf
- Andrade, R. (2000) "Applying Use Case Maps and Formal Methods to the Development of Wireless Mobile ATM Networks". In: *Lfm2000, The Fifth NASA Langley Formal Methods Workshop*, Williamsburg, Virginia, USA, June 2000. http://www.UseCaseMaps.org/pub/lfm2000.pdf
- Bordeleau, F. and Buhr, R.J.A. (1997) "The UCM-ROOM Design Method: from Use Case Maps to Communicating State Machines". In: *Conference on the Engineering of Computer-Based Systems*, Monterey, USA, March 1997. http://www.UseCaseMaps.org/pub/UCM-ROOM.pdf
- Bordeleau, F. (1999) *A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical Finite State Machines*. Ph.D. thesis, School of Computer Science, Carleton University, Ottawa, Canada. http://www.UseCaseMaps.org/pub/fb_phdthesis.pdf
- Bordeleau, F. and Cameron, D. (2000) "On the Relationship between Use Case Maps and Message Sequence Charts". In: *2nd Workshop of the SDL Forum Society on SDL and MSC (SAM2000)*, Grenoble, France, June 2000. http://www.UseCaseMaps.org/pub/sam2000.pdf
- Buhr, R.J.A. and Casselman, R.S. (1996) *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, USA. http://www.UseCaseMaps.org/pub/UCM_book95.pdf
- Buhr, R.J.A. (1998) "Use Case Maps as Architectural Entities for Complex Systems". In: *IEEE Transactions on Software Engineering, Special Issue on Scenario Management*. Vol. 24, No. 12, December 1998, 1131-1155. http://www.UseCaseMaps.org/pub/tse98final.pdf
- Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., and Mankovski, S. (1998) "High Level, Multi-agent Prototypes from a Scenario-Path Notation: A Feature-Interaction Example". In: H.S. Nwana and D.T. Ndumu (Eds), *PAAM'98, Third Conference on Practical Application of Intelligent Agents and Multi-Agents, London*, UK, March 1998, 277-295. http://www.UseCaseMaps.org/pub/4paam98.pdf
- Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., and Mankovski, S. (1998) "Feature-Interaction Visualization and Resolution in an Agent Environment". In: K. Kimbler and L. G. Bouma (Eds), *Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98)*, Lund, Sweden, September 1998. IOS Press, 135-149. http://www.UseCaseMaps.org/pub/fiw98.pdf
- Buhr, R.J.A., Elammari, M., Gray, T., and Mankovski, S. (1998) "Applying Use Case Maps to Multi-agent Systems: A Feature Interaction Example". In: *Hawaii International Conference on System Sciences (HICSS'98)*, Hawaii, January 1998. http://www.useCaseMaps.org/pub/hiccs98.pdf
- Buhr, R.J.A. (1999), "Understanding Macroscopic Behaviour Patterns in Object-Oriented Frameworks, with Use Case Maps". In: Fayad, M.E., Schmidt, D.C., and Johnson, R.E. (eds) *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley & Sons. http://www.UseCaseMaps.org/pub/uoof.pdf
- Hodges, J. and Visser, J. (1999) "Accelerating Wireless Intelligent Network Standards Through Formal Techniques". In: *IEEE 1999 Vehicular Technology Conference (VTC'99)*, Houston (TX), USA. http://www.UseCaseMaps.org/pub/vtc99.pdf
- Miga, A. (1998) Application of Use Case Maps to System Design with Tool Support. M.Eng. thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada. http://www.UseCaseMaps.org/pub/am_thesis.pdf
- Nakamura, M., Kikuno, T., Hassine, J., and Logrippo, L. (2000). "Feature Interaction Filtering with Use Case Maps at Requirements Stage". In: *Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00)*, Glasgow, Scotland, UK, May 2000. http://www.UseCaseMaps.org/pub/fiw00filter.pdf

- Sales, I. and Probert, R. (2000) "From High-Level Behaviour to High-Level Design: Use Case Maps to Specification and Description Language". In: *SBRC'2000, 18º Simpósio Brasileiro de Redes de Computadores*, Belo Horizonte, Brazil. http://www.UseCaseMaps.org/pub/sbrc00.pdf
- Scratchley, W.C. and Woodside, C.M. (1999) "Evaluating Concurrency Options in Software Specifications". In: *MASCOTS'99, Seventh International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, College Park, MD, USA, October 1999, 330-338. http://www.UseCaseMaps.org/pub/mascots99.pdf
- Scratchley, W.C. (2000) *Evaluation and Diagnosis of Concurrency Architectures*. Ph.D. thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, June 2000.
- Smith, C.U. (1990) *Performance Engineering of Software Systems.* Addison-Wesley.
- *Use Case Maps Web Page* and *UCM User Group* (1999). http://www.UseCaseMaps.org
- Yi, Z. (2000) *CNAP Specification and Validation: A Design Methodology Using LOTOS and UCM*. M.Sc. thesis, SITE, University of Ottawa, Canada. http://lotos.site.uottawa.ca/ftp/pub/Lotos/Theses/zm_msc.pdf


## III.3 Usability

This section contains a list of Web sites, books, and standards relating to usability.


- Dumas, J.S. & Redish, J.C. (1993). *A Practical Guide to Usability Testing*. Norwood, USA. Ablex.
- ISO 9241-11:1998, *Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11 Guidance on Usability*.
- ISO 13407:1999, *Human-centred design processes for interactive systems*. Technical Committee/Sub-Committee: TC159/SC4. Geneva.
- ISO 14598-5:1998 *Information Technology Software Product Evaluation – Part 5: Process for Evaluation*. Technical Committee/Sub-Committee: JT1/SC7. Geneva.
- Nielsen, J. (1993). *Usability engineering*. San Francisco, USA. Morgan Kaufmann.
- Rubin, J. (1994). *Handbook of Usability Testing. How to plan, design, and conduct effective tests*. New York, USA. Wiley.
- U.S. National Institute of Standards and Technology — NIST (1997), *Industry Usability Reporting (IUSR)*, http://zing.ncsl.nist.gov/iusr/overview.html
- U.S. National Institute of Standards and Technology — NIST (1999), *Common Industry Format for Usability Test Reports*, version 1.1, October 28, 1999. http://zing.ncsl.nist.gov/iusr/documents/cifv1.1b.htm

# Appendix IV
# Guidelines for the maintenance of URN

## IV.1 Maintenance of URN

This section describes the terminology and rules for maintenance of URN agreed at the Study Group 10 meeting in November 2000, and the associated "change request procedure".

Terminology:

a) An *error* is an internal inconsistency within Z.URN.

b) A *textual correction* is a change to text or diagrams of Z.URN that corrects clerical or typographical errors.

c) An *open item* is a concern identified but not resolved. An open item may be identified either by a Change Request, or by agreement of the Study Group or Working Party.

d) A *deficiency* is an issue identified where the semantics of URN are not (clearly) defined by Z.URN.

e) A *clarification* is a change to the text or diagrams of Z.URN, which clarifies previous text or diagrams that could be ambiguously understood without the clarification. The clarification should attempt to make Z.URN correspond to the semantics of URN as understood by the Study Group or Working Party.

f) A *modification* is a change to the text or diagrams of Z.URN that changes the semantics of URN.

g) A *deprecated feature* is a feature of URN that is to be removed from URN in the next revision of Z.URN.

h) An *extension* is a new feature, which must not change the semantics of features defined in Z.URN.

## IV.2 Rules for maintenance

In the following text references to Z.URN shall be considered to include Annexes, Appendices, and Supplements, as well as any Addendum, or Amendment or Corrigendum or Implementers Guide.

a) When an error or deficiency is detected in Z.URN, it must be corrected or clarified. The correction of an error should imply as small a change as possible. Error corrections and clarifications will be put into the Master list of Changes for Z.URN and come into effect immediately.

b) Except for error corrections and resolution of open items from the previous study period, modifications and extensions should only be considered as the result of a request for change that is supported by a substantial user community. A request for change should be followed by investigation by the Study Group or Working Party in collaboration with representatives of the user group, so that the need and benefit are clearly established and it is certain that an existing feature of URN is unsuitable.

c) Modifications and extensions not resulting from error correction shall be widely publicised and the views of users and toolmakers canvassed before the change is adopted. Unless there are special circumstances requiring such changes to be implemented as soon as possible, such changes will not be recommended until Z.URN.

d) Until a revised Z.URN is published a Master list of Changes to Z.URN will be maintained covering Z.URN and all annexes except the formal definition. Appendices, Addenda, Corrigenda, Implementers' guides or Supplements will be issued as decided by the Study Group. To ensure effective distribution of the Master list of Changes to Z.URN, it will be published as COM Reports and by appropriate electronic means.

e) For deficiencies in Z.URN the formal definition should be consulted. This may lead to either a clarification or correction that is recorded in the Master list of changes to Z.URN.

## IV.3 Change request procedure

The change request procedure is designed to enable URN users from within and outside ITU-T to ask questions about the precise meaning of Z.URN, make suggestions for changes to URN or Z.URN, and to provide feedback on proposed changes to URN. The URN experts' group shall publish proposed changes to URN before they are implemented.

Requests for changes should either use the Change Request Form (see below) or provide the information listed by the form. The kind of request should be clearly indicated (error correction, clarification, simplification, extension, modification or deprecated feature). It is also important that for any change other than an error correction, the amount of user support for the request is indicated.

The ITU-T Study Group responsible for Z.URN should formally process all change requests at scheduled meetings. For corrections or clarifications the changes may be put on the list of corrections without consulting users. Otherwise a list of open items is compiled. The information should be distributed to users:

- as ITU-T white contribution reports;
- as electronic mail to URN mailing lists (such as ITU-T informal list, and URNnews@URN-forum.org);
- by others means as agreed by the Study Group 10 experts.

Study Group experts should determine the level of support and opposition for each change and evaluate reactions from users. A change will only be put on the accepted list of changes if there is substantial user support and no serious objections to the proposal from more than just a few users. Finally all accepted changes will be incorporated into a revised Z.URN. Users should be aware that until changes have been incorporated and approved by Study Group responsible for Z.URN they are not recommended by ITU-T.

# URN Change Request Form

| | | |
|---|---|---|
| Please fill in the following details | | |
| Character of change: | ❑ error correction | ❑ clarification |
| | ❑ simplification | ❑ extension |
| | ❑ modification | ❑ decommission |
| Short summary of 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.URN) Rapporteur, c/o ITU-T, Place des Nations, CH-1211, Geneva 20, Switzerland. Fax: +41 22 730 5853, e-mail: URN.rapporteur@itu.int