

# **Textual User Requirements Notation**

Ruchika Kumar

Master of Engineering

Department of Electrical and Computer Engineering  
McGill University, Montreal

July 2018

A thesis submitted to McGill University in partial fulfillment of the requirements of  
the degree of Master of Engineering

© Ruchika Kumar 2018

## **ABSTRACT**

This thesis describes the textual syntax for the User Requirements Notation (URN), called TURN (Textual URN), and its transformation into the already existing graphical representation of URN. The main objective of TURN is to support the modeling of very large URN specifications where thousands of separate graphs or maps as distinct diagrams become unwieldy to navigate. In addition, the entering of large specifications in graphical tools has proven tedious, as the modeler must be concerned with layout issues that are unrelated to the information that is attempted to be modeled. In general, TURN offers an alternative input medium for URN specifications which is easier, faster and more scalable.

The TURN grammar strikes a balance between supporting as many language features of URN as possible and the usability, convenience, and expediency of the textual syntax. The textual syntax for the Goal-oriented Requirement Language (GRL) covers the abstract grammar of GRL in its entirety. The textual syntax for Use Case Maps (UCM) does not cover performance concepts, component types, empty points and scenario definitions from the abstract grammar of UCM but everything else from the abstract grammar of UCM. The first two are not essential to a URN specification, while empty points are irrelevant for a textual syntax and scenario definitions are left for future work.

The metamodels for URN and TURN are presented and a comparison is shown between the two, followed by the TURN specification. A transformation from TURN to URN is also described along with the tools and methodology. Moreover, a detailed validation is performed by

an exhaustive execution of test cases for the TURN specifications and the implementation of their transformation to the graphical URN representation.

## ABRÉGÉ

Cette thèse décrit la syntaxe textuelle de la Notation des prescriptions utilisateur (URN), appelée TURN (Textual User Requirements Notation), et sa transformation dans la représentation graphique déjà existante de URN. L'objectif principal de TURN est de soutenir la modélisation de très grandes spécifications URN où des milliers de diagrammes divers et distincts deviennent difficiles à naviguer. De plus, la création de grandes spécifications dans les outils graphiques s'est avérée problématique, car le modélisateur doit se préoccuper de difficultés superflues comme la mise en page des éléments, qui ne sont pas liées aux informations que l'on tente de modéliser. En général, TURN offre un point d'entrée alternatif pour les spécifications URN qui est plus facile, plus rapide et plus extensible.

La grammaire TURN tente d'établir un équilibre entre la prise en charge du plus grand nombre possible des fonctionnalités offertes par URN et la facilité d'utilisation, la simplicité et le pouvoir expressif de sa syntaxe textuelle. La syntaxe textuelle du Goal-oriented Requirement Language (GRL) couvre la grammaire abstraite de GRL dans son intégralité. La syntaxe textuelle de la notation Use Case Map (UCM) ne couvre pas les concepts de performance, les types de composantes, les points vides et les définitions de scénarios de la grammaire abstraite d'UCM, mais supporte le reste de la grammaire abstraite d'UCM. Les deux premiers points susmentionnés, soit les concepts de performance et les types de composantes, ne sont pas essentiels pour une spécification URN, tandis que les points vides ne sont pertinents que pour sa syntaxe graphique. Finalement, les définitions de scénarios sont laissées pour des travaux futurs.

Les métamodèles pour URN et TURN sont présentés et une comparaison entre les deux est faite. Une transformation de TURN vers URN est également décrite avec ses outils et sa méthodologie. De plus, une validation détaillée est effectuée par une exécution exhaustive des cas de tests pour les spécifications TURN et la mise en œuvre de leur transformation en représentation graphique URN.

## **ACKNOWLEDGMENTS**

I would like to use this opportunity to express my gratitude to everyone who supported me throughout my Masters study.

Firstly, I would like to sincerely thank my supervisor Professor Gunter Mussbacher for his continuous support, patience and motivation throughout my thesis. I always found his door open to my questions about research and writing, and received assistance and dedicated involvement throughout the process. I could not have thought to get a better supervisor than him.

Besides my supervisor, I would like to thank the rest of the people who contributed to my thesis in some way. My sincere thanks go to Prof. Jörg Kienzle and my mates from Software Engineering lab for providing me with valuable feedback during my thesis. Their insightful comments always helped me improve my work and widen my research area from various perspectives.

I am grateful to the funding source - NSERC that helped me pursue my graduate school studies without being concerned about the tuition and living expenses involved.

I express my warm thanks to all my friends here in Montreal, who always supported me and made these two years great fun.

I am also grateful to the Eclipse Community Forums and their support team for helping me understand the technical concepts and find solutions to my problems.

And last but not the least, a very special gratitude goes to my parents and my elder brother for providing me with continuous moral support and encouragement throughout my years of study. This work would not have been possible without their motivation. Thank you.

## TABLE OF CONTENTS

ABSTRACT.....	1
ABRÉGÉ.....	3
ACKNOWLEDGMENTS .....	5
TABLE OF CONTENTS.....	7
LIST OF TABLES .....	9
LIST OF FIGURES .....	10
LIST OF LISTINGS .....	11
CHAPTER 1: Introduction .....	12
1.1 Motivation.....	12
1.2 Contributions and Methodology .....	14
1.3 Abbreviations .....	14
1.4 Thesis Organization .....	15
CHAPTER 2: Background Information.....	17
2.1 User Requirements Notation.....	17
2.1.1 GRL Notation.....	17
2.1.2 UCM Notation .....	20
2.2 URN Metamodel .....	23
2.3 Summary .....	31
CHAPTER 3: Textual URN Specification .....	32
3.1 Differences to URN Metamodel .....	32
3.2 TURN Metamodel .....	36
3.3 Examples of Concrete Syntax .....	45
3.4 Introduction to Xtext Framework & TURN Grammar .....	52
3.4.1 Xtext Framework .....	52
3.4.2 TURN Grammar .....	59
3.5 Summary .....	60
CHAPTER 4: TURN to URN Transformation .....	61
4.1 Backward Links .....	61
4.2 Transformation Rules.....	62
4.3 Summary .....	68
CHAPTER 5: Validation .....	69
5.1 Implementation .....	69
5.1.1 ATL (Atlas Transformation Language) Tool .....	69
5.1.2 jUCMNav Tool .....	71
5.1.3 Implementation Methodology.....	72
5.1.4 Phase 0 .....	72
5.1.5 Phase 1 .....	72
5.1.6 Phase 2 .....	73
5.2 Test Cases .....	77
5.3 Summary .....	80



CHAPTER 6: Related Work .....	81
CHAPTER 7: Conclusions and Future Work .....	84
REFERENCES .....	87
Appendix A.....	90
Appendix B.....	96

## LIST OF TABLES

Table 1 Transformation for URNSpec and ConcreteURNSpec.....	62
Table 2: Transformation for Metadata, Concern and URNLink.....	63
Table 3: Transformation for Actors, IntentionalElements and ElementLinks.....	63
Table 4: Transformation for Strategies and ContributionContexts.....	64
Table 5: Transformation for UCMmap.....	65
Table 6: Transformation for StartPoint and RespRef .....	65
Table 7: Transformation for remaining path nodes .....	66
Table 8: Test cases for Responsibility followed by a PathBodyNode or a RegularEnd.....	78
Table 9: Test cases for Responsibility followed by a ReferencedEnd.....	79
Table 10: Test cases for full specification of Responsibility.....	80

## LIST OF FIGURES

Figure 1: Excerpt of GRL Model [17] .....	19
Figure 2: Examples of UCMs [17].....	22
Figure 3: URN Main [38] .....	23
Figure 4: URN Core [38] .....	24
Figure 5: GRL Core [38].....	25
Figure 6: GRL Links [38] .....	25
Figure 7: GRL Strategy [38] .....	26
Figure 8: GRL Overview [38].....	27
Figure 9: UCM Map Links [38].....	28
Figure 10: UCM PathNodes [38].....	29
Figure 11: UCM PluginBindings [38] .....	30
Figure 12: URN Concern [38] .....	30
Figure 13: Overview of Main Differences to GRL Metamodel .....	33
Figure 14: Overview of Main Differences to UCM Metamodel .....	35
Figure 15: TURN Main.....	36
Figure 16: TGRL Intentional Elements and Element Links .....	37
Figure 17: TGRL Contribution Context .....	38
Figure 18: TGRL Strategies and Evaluation.....	38
Figure 19: Key Difference Between UCM and TUCM – Paths .....	39
Figure 20: Key Difference Between UCM and TUCM – Implicit OrJoins.....	40
Figure 21: TUCM Path .....	41
Figure 22: TUCM Map .....	42
Figure 23: TUCM OrFork and AndFork .....	42
Figure 24: TUCM Stub and Bindings.....	43
Figure 25: Connecting Path Bodies in OrFork, AndFork and Stub .....	44
Figure 26: TUCM Connects.....	44
Figure 27: TUCM ComponentRefs .....	45
Figure 28: Metamodel corresponding to Xtext features example in Listing 7 and Listing 8 .....	56
Figure 29: No Backlink from ElementLink to IntentionalElement .....	61
Figure 30: Backlink from ElementLink to IntentionalElement .....	62
Figure 31: Implementation methodology.....	72
Figure 32: Map TL after Phase 1 .....	74
Figure 33: Map TL after binding elements to Components in Phase 2 .....	74
Figure 34: Map SC after Phase 1 .....	75
Figure 35: Map SC after Phase 2 (with an OrJoin).....	76
Figure 36: Map TL after Phase 2 .....	76
Figure 37: Map with asynchronous Connect after Phase 1.....	77
Figure 38: Map with asynchronous Connect after Phase 2.....	77

## LIST OF LISTINGS

Listing 1: Example TURN for URN Top Level .....	46
Listing 2: Example TURN for GRL Core.....	46
Listing 3: Example TURN for GRL Strategies and Contribution Changes .....	48
Listing 4: Example TURN for UCM .....	49
Listing 5: Example Data type rule .....	53
Listing 6: Example Value Converter for Positive Integer .....	53
Listing 7: Xtext features example .....	55
Listing 8: Unassigned Rule Call example.....	56
Listing 9: Linking example.....	57
Listing 10: Rule for defining QualifiedName for an IntentionalElement.....	57
Listing 11: TURN scoping mechanism.....	58
Listing 12: TURN scoping mechanism.....	58
Listing 13: TURN validation for AndFork .....	59
Listing 14: Matched Rule example .....	70
Listing 15: Called Rule example.....	71
Listing 16: Helper Rule example .....	71

## CHAPTER 1: Introduction

The User Requirements Notation (URN) [4] is a lightweight graphical language intended for modeling and analyzing requirements in the form of goals and scenarios and the links between them. It consists of two notations – the Goal-oriented Requirement Language (GRL) and Use Case Maps (UCMs). URN models can be used to discover, specify and analyze requirements for their correctness and completeness. Currently, URN supports only a concrete graphical syntax and we propose a textual syntax for URN (TURN) as it has been proved to provide many benefits such as improving the analyzability and modifiability of the model [16], have a shorter learning curve [26] and efficient consistency checking [21].

Since different modelers may have different preferences of the two concrete representations, there should be a way of generating one concrete syntax from another to achieve consistency in models. Hence, a process of automation is required when a model written in one concrete syntax needs to be viewed in a different concrete syntax. Normally, the same abstract syntax is used for different concrete representations, but in the case of TURN a slightly different approach was used as explained in the following section.

### ***1.1 Motivation***

URN currently supports only a graphical representation using the jUCMNav tool [23]. A graphical representation has some advantages like it is easier to grasp and requires less mental effort [27]. However, there exist some disadvantages too for the currently dominating graphical based approach.

Several frameworks and tools now exist that enable us to create concrete textual syntaxes for models. A textual syntax for URN is beneficial as it promotes usability, productivity and scalability of URN models. The entering of large specifications in graphical tools has proven tedious, as the modeler must be concerned with layout issues that are unrelated to the information that is attempted to be modeled. Other advantages of textual notation include fast editing style, usage of error markers, providing autocompletion and quick fixes. Furthermore, they can easily be integrated into existing tools such as diff/merge or information interchange through e-mail, or blogs.

Normally, two different concrete representations require the same metamodel but in this case, we have two metamodels for URN and TURN. The metamodel derived from textual syntax (TURN) is a little different from the URN metamodel as the URN metamodel contains a few concepts that are needed only for a graphical syntax and not for textual syntax, e.g., node connections and node labels. Furthermore, the TURN metamodel requires the explicit notion of a path, which does not exist as a first-class concept in the URN metamodel. One advantage of having a different metamodel for the textual syntax is that we are less constrained in fully exploiting existing frameworks for textual languages such as Xtext [40] which automatically creates a metamodel, parser and editor from a grammar definition. There is also a disadvantage because the two different choices of representations need to be automatically synchronized to ensure consistency. This automation can be achieved by performing a transformation of one representation into another using a transformation tool and fortunately requires reasonable effort since the two metamodels are fairly similar.

## ***1.2 Contributions and Methodology***

In order to solve the problem stated in the previous section, a textual grammar for URN (TURN) is defined using the Xtext Framework. Xtext can then automatically generate an editor where a model can be specified. This model is further saved into XMI form and transformed into .jucm file using the transformation rules in ATL tool.

The definition of textual grammar using Xtext and transformation of TURN model into URN model using ATL tool are implemented as a proof a concept. In order to validate this correct implementation of the textual language and its transformation to graphical URN models, several test cases are written and executed. At least one test case exists for each tuple existing of one GRL source node (out of 6 types of GRL nodes), one GRL target node (again out of 6 types of GRL nodes) and one GRL link (out of 3 types of GRL links) connecting the source and target node. Furthermore, at least one test case exists for each pair of UCM path elements (out of 12 types of UCM path elements), where the second element in the pair immediately follows the first element in the UCM specification. Each tuple is specified by the Xtext editor in a basic textual GRL or UCM specification and then transformed to a graphical URN model. In addition, several test cases check whether invalid textual specifications are rejected by the Xtext editor.

## ***1.3 Abbreviations***

The following abbreviations are used in this thesis document:

- AST: Abstract Syntax Tree
- ATL: Atlas Transformation Language
- BNF: Backus Naur Form
- DSL: Domain Specific Language
- EMF: Eclipse Modeling Framework
- EOL: Epsilon Object Language
- ETL: Epsilon Transformation Language

- GRL: Goal-oriented Requirement Language
- HUTN: Human-Usable Textual Notation
- ITU-T: International Telecommunication Union – Telecommunication Standardization Sector
- JTL: Janus Transformation Language
- KPI: Key Performance Indicator
- M2M: Model-to-Model
- MOF: Meta-Object Facility
- MSC: Message Sequence Chart
- OCL: Object Constraint Language
- OMG: Object Management Group
- QVT: Query/View/Transformation
- SC: Simple Connection
- SDL: Specification and Description Language
- TCS: Terminating Call Screening
- TGRL: Textual Goal-Oriented Requirement Language
- TL: Teen Line
- TTCN-3: Testing and Test Control Notation
- TUCM: Textual Use Case Maps
- TURN: Textual User Requirements Notation
- UCM: Use Case Maps
- UML: Unified Modeling Language
- URN: User Requirements Notation
- XMI: XML Metadata Interchange

## ***1.4 Thesis Organization***

The remainder of this thesis document is organized as follows:

Chapter 2 gives the background information on URN and presents its metamodel.

Chapter 3 describes the Textual URN concepts. It explains the differences between the Textual URN and URN metamodel while also presenting the Textual URN metamodel. A few examples of concrete syntax are also illustrated followed by a brief introduction to the Xtext framework used to describe the TURN grammar.

Chapter 4 describes the transformation rule written for each URN element in ATL.



Chapter 5 describes the ATL and jUCMNav tools that are used for the TURN-URN transformation, implementation methodology, the two phases of transformation, and execution of test cases in order to validate the textual syntax and their transformation to the already existing graphical syntax.

Chapter 6 gives a brief overview of related work.

Chapter 7 concludes the thesis and presents future work.

Appendix A presents the TURN grammar.

Appendix B presents the test cases written and executed for validating the TURN specification and its transformation to URN.

## **CHAPTER 2: Background Information**

This chapter provides a conceptual background about the User Requirements Notation (URN) and its metamodel, before introducing the Textual User Requirements Notation (TURN) in following chapters.

### ***2.1 User Requirements Notation***

The User Requirements Notation (URN) [4] is a graphical language intended for elicitation, analysis, specification and validation of requirements. It combines two complementary notations: the Goal-oriented Requirement Language (GRL) [17] for modeling goal-oriented and intentional concepts, i.e., non-functional requirements; and Use Case Maps (UCMs) [17] for functional and operational requirements. It is the first and currently only standard which explicitly addresses goals (non-functional requirements with GRL) in addition to scenarios (functional requirements with UCMs) in a graphical way in one unified language. It is standardized by the International Telecommunication Union in the Z.15x series [17].

URN models can be used to specify and analyze various types of reactive systems, business processes and goals of organizations, and telecommunications standards. It provides insight at the requirements level that enables designers to reason about feature interactions and performance trade-offs early in the design process. A model based on the URN metamodel is able to specify everything that a textual syntax of URN can specify.

#### **2.1.1 GRL Notation**

Goal modeling has been recognized as an effective approach to Requirements Engineering [29]. It allows the modelers to visualize and communicate common concepts of goal

modeling notations such as stakeholders and their priorities, alternative solutions, decisions and rationale that impact these priorities. It can also capture dependencies between various stakeholders. Several goal modeling frameworks exist besides GRL such as i\* [41], the NFR framework [8], KAOS [30], and Tropos [7].

A GRL goal model shows the high-level business goals and non-functional requirements of interest to a stakeholder and the alternative solutions that can accomplish these goals and high-level requirements. A stakeholder of a system is represented as an *Actor* ( $\odot$ , e.g., “Telecom Provider” in Figure 1 and corresponding class in the GRL metamodel in Figure 6). An actor holds the intentions, i.e., the actor wants goals to be achieved, tasks to be performed, resources to be available and softgoals to be satisfied. Various intentional elements (softgoal, goal, task, resource) are available to capture the mentioned concerns. Objectives and qualities are modeled with softgoals and goals. A *Softgoal* ( $\sqsubset$ , e.g. High Reliability in Figure 1, and a type of IntentionalElement as indicated in Figure 8) is used to represent objectives that have no definite measure of satisfaction, whereas a *Goal* ( $\sqsubset$ , e.g., “Voice Connection Be Setup” in Figure 1, and a type of IntentionalElement in Figure 8) is used when the objective is clear and quantifiable. Softgoals are related to non-functional requirements, whereas goals are related to functional requirements and measurable non-functional requirements. A *Task* ( $\sqsubset$ , e.g., “Make Voice Connection Over Wireless” in Figure 1, a type of IntentionalElement as indicated in Figure 8) is the proposed solution that achieves a goal or satisfies a softgoal. A goal model may also document facts or *Beliefs* ( $\odot$ , e.g., “Wireless is less reliable than Internet” in Figure 1, and corresponding class in GRL metamodel in Figure 8) to capture the rationale. Softgoals, goals and tasks may require *Resources* ( $\square$ , e.g., “Logging Equipment” in Figure 1, and a type of

IntentionalElement in Figure 8) to be achieved or completed. *Key Performance Indicators* (KPIs) (e.g., “Failure Rate for Voice Connection Over Internet” in Figure 1) allow real-life measured values to be integrated into the analysis of a goal model, hence improving the accuracy of the analysis.

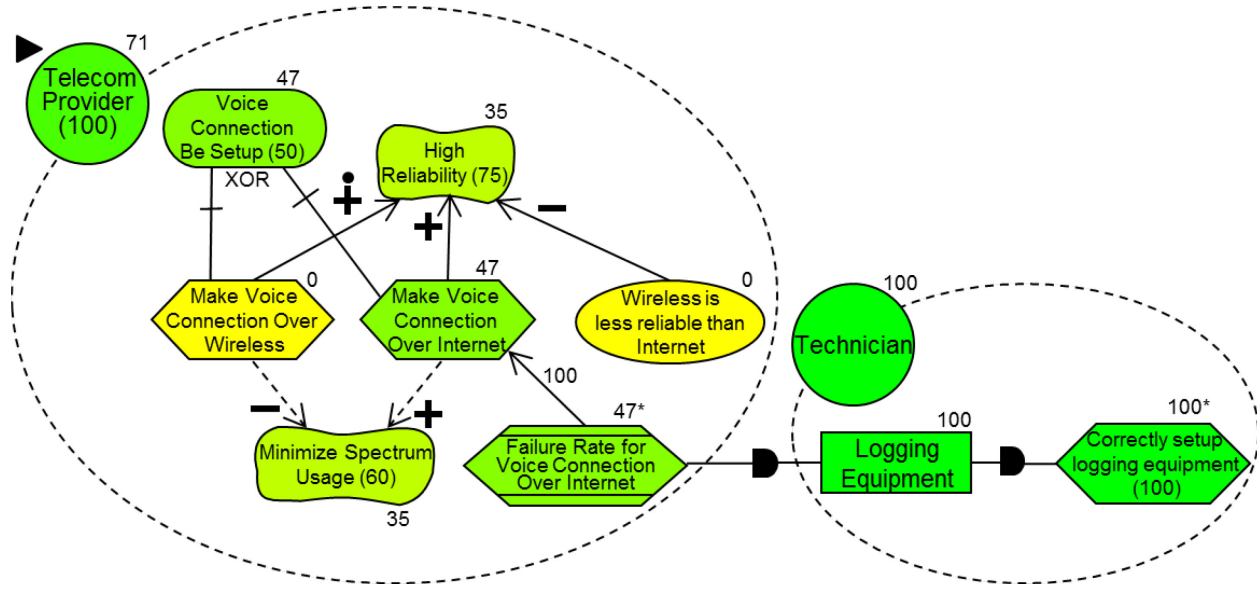


Figure 1: Excerpt of GRL Model [17]

Element links are used to connect various elements in a requirement model using structural and intentional relationships. *Decomposition* links ( $+ \text{---}$ , e.g., between “Voice Connection Be Setup” and “Make Voice Connection Over Wireless” in Figure 1, and corresponding class in Figure 6) allow an element to be decomposed into sub-elements. GRL supports three variants of decompositions – AND, XOR, and IOR (see *DecompositionType* in Figure 6). A *Dependency* link ( $- \text{---}$ , e.g., between “Failure Rate for Voice Connection Over Internet” and “Logging Equipment” in Figure 1, and corresponding class in Figure 6) is used to model dependency of one element on another element, typically across actor boundaries. Desired influences of one element on another are represented using a *Contribution* link ( $\rightarrow$ , e.g., between

“Make Voice Connection Over Internet” and “High Reliability” in Figure 1, and corresponding class in Figure 6). Contributions can be qualitative (e.g., + or – in Figure 1; i.e., from Make to Break, see ContributionType in Figure 6) or quantitative (integer value between –100 and 100). An extension to the URN standard allows quantitative contribution from an element also to be specified relative to other contributions of the same element in order to allow local comparison of all the elements in the goal model [10][11].

### **2.1.2 UCM Notation**

Use Case modeling is used to bridge the gap between abstract and informal descriptions that are useful at early stages of system development and formal and concrete descriptions that are useful at later stages of system development. A Use Case model allows for reasoning about scenario descriptions and is mainly used for operational requirements, functional requirements and business processes. Several scenario notations [3] exist and offer capabilities such as describing scenarios as first-class entities without requiring reference to system sub-components or sub-component states. In URN, scenarios are described with Use Case Maps.

A Use Case Map (UCM) models causal relationships involving concurrency and partial ordering of steps in a scenario. A UCM links causes to effects and abstract from the details of component interactions that are generally expressed as message sequences. Moreover, UCMs provide their users with the ability to dynamically refine capabilities for variations of scenarios and structures, and they allow incremental development and integration of complex scenarios. A UCM can also be transformed into sequence diagrams/MSCs, performance models and test cases.

In UCMs, a *Scenario* is used to partially describe a system usage that is defined as a set of partially-ordered responsibilities performed by the system to transform inputs to outputs while satisfying preconditions and postconditions. UCM *Responsibilities* (x, e.g., “forwardSignal” in Simple Connection UCM in Figure 2, and the corresponding metamodel class in Figure 9) are activities that represent something to be performed (operation, action, task, function, etc.), i.e., the steps in a scenario. A responsibility can also be associated or allocated to a component. A *Component* (□, e.g., Originating User in Figure 2, and the corresponding metamodel class in Figure 9) can represent software entities like objects, processes, databases etc. as well as non-software entities like actors and hardware. A *Start Point* (●, e.g., “request” in Figure 2, and the corresponding metamodel class in Figure 10) captures pre-conditions and triggering events while an *End Point* (■, e.g., “busy” in Figure 2, and the corresponding metamodel class in Figure 10) captures resulting events and post-conditions. Scenarios progress along paths from start points to end points that can also support responsibilities, alternatives (OR-fork), and concurrent behavior (AND-fork) and may also join (OR-join and AND-join). An *OR-join* (see corresponding metamodel class in Figure 10) indicates overlapping of scenarios that share common paths while an *AND-join* (see corresponding metamodel class in Figure 10) synchronizes two or more paths that must have been traversed for the scenario to progress. An alternative branch is guided by a *Condition* (“[!busy]” in Terminating Features UCM in Figure 2) that needs to be true for the guarded path to be followed. A *Stub* (see corresponding metamodel class in Figure 10) is a container for sub-maps or plugin maps. Any map can be a plugin with its start points and end points connected to identifiable input and output segments of a stub. This binding relationship (see PluginBinding in Figure 11) ensures that the paths flow from parent maps to plugin maps, and back to parent maps. Two variants of stubs exist – a *Static Stub* (◇, e.g., “Originating” in

Simple Connection UCM in Figure 2) that can contain only one plugin map and a *Dynamic Stub* (◇, e.g., OrigFeatures in Originating Features UCM in Figure 2) that can contain more than one plugin map with pre-conditions used to choose the correct map(s) for run-time traversal. In UCM, waiting can be done using *Waiting Places* and *Timers* (⊖, e.g., getPIN in Teen Line UCM in Figure 2, and corresponding metamodel classes in Figure 10). The waiting period ends when there is an event received by the waiting place or the timer from the environment or another scenario. A timer also has a timeout path (⚡, e.g., the path from the timer to the deny responsibility in Figure 2) that can be followed when it does not receive a trigger in time.

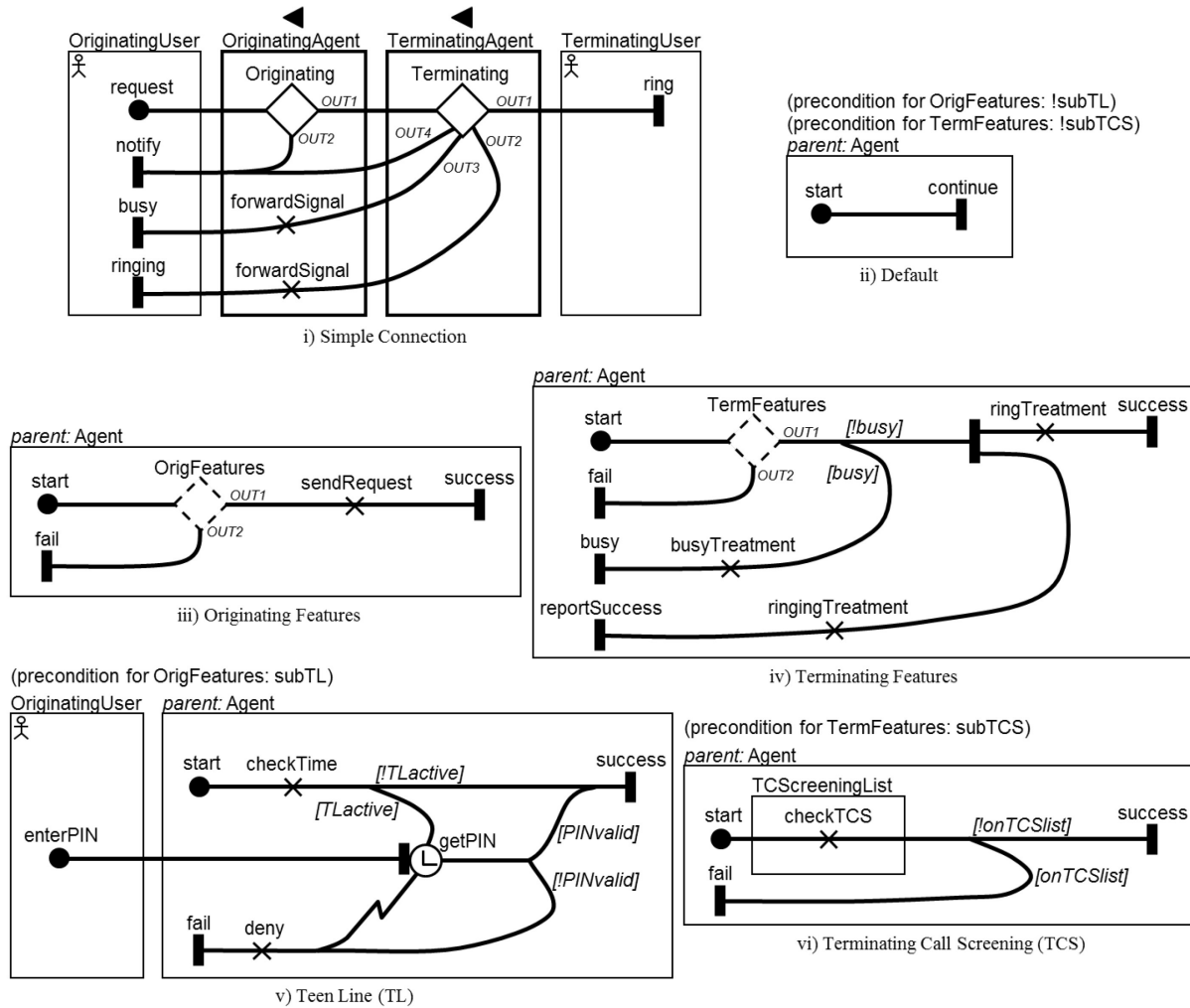


Figure 2: Examples of UCMs [17]

URN allows typed links to be defined between model elements for traceability reasons. Typically, GRL model elements such as actors and tasks are connected to UCM model elements such as components and responsibility or stubs. Outgoing URN links are indicated by a triangle pointing to the right (►, e.g., URN link for actor “Telecom Provider” in Figure 1). Incoming URN links are indicated by a triangle pointing to the left (◄, e.g., URN links for components “OriginatingAgent” and “TerminatingAgent” in Figure 2).

## 2.2 URN Metamodel

The URN metamodel has a root container called *URNspec* that contains a *grlspec*, a *ucmspec*, a *urndef* and *urnLinks*. Each *URNlink* is connected to a source *URNmodelElement* and a target *URNmodelElement*. Furthermore, a *Metadata* is a name-value pair used to attach arbitrary information to a *URNspec*, *URNlink* or *URNmodelElement* in the URN model (see Figure 3). *GRLspec* serves as a container for GRL specification elements while *UCMspec* is its counterpart for UCM specification elements.

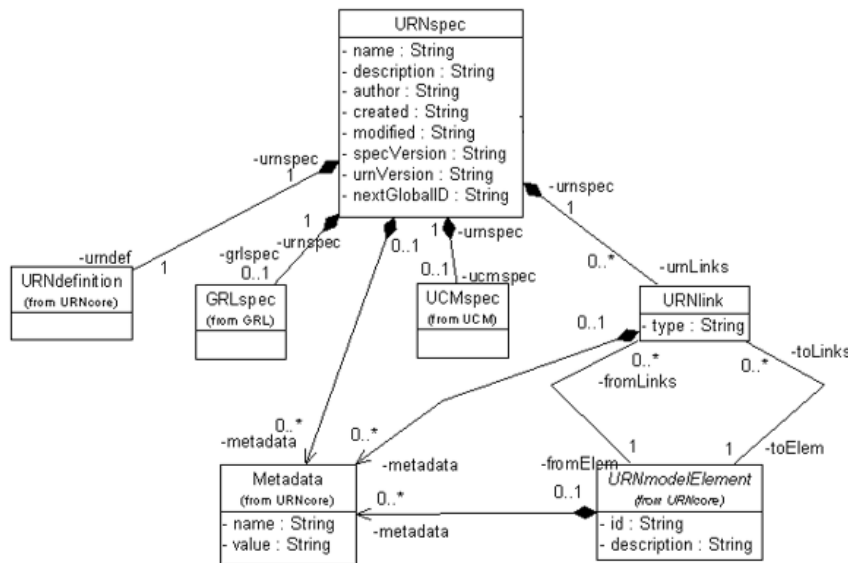


Figure 3: URN Main [38]



A *URNdefinition* consists of specDiagrams (which could be either of type *GRLGraph* or *UCMMap*), responsibilities, componentTypes and components (see Figure 4). A *ComponentType* allows for user-defined types of components.

The *IURNDiagram* can be a *GRLGraph* for GRL elements or a *UCMmap* for UCM elements. A *GRLGraph* consists of all the references to GRL nodes, i.e., *CollapsedActorRefs*, *Beliefs* and *IntentionalElementRefs*; and their connections, i.e., *BeliefLinks* and *LinkRefs*; and container references, i.e., *ActorRefs*. Each of the references is connected to its node type (e.g., a *LinkRef* has one *ElementLink* connected to it) (see Figure 5). A *CollapsedActorRef* is different from an *ActorRef* in that it is used to highlight dependencies among actors without showing the internal goal hierarchy of the actor.

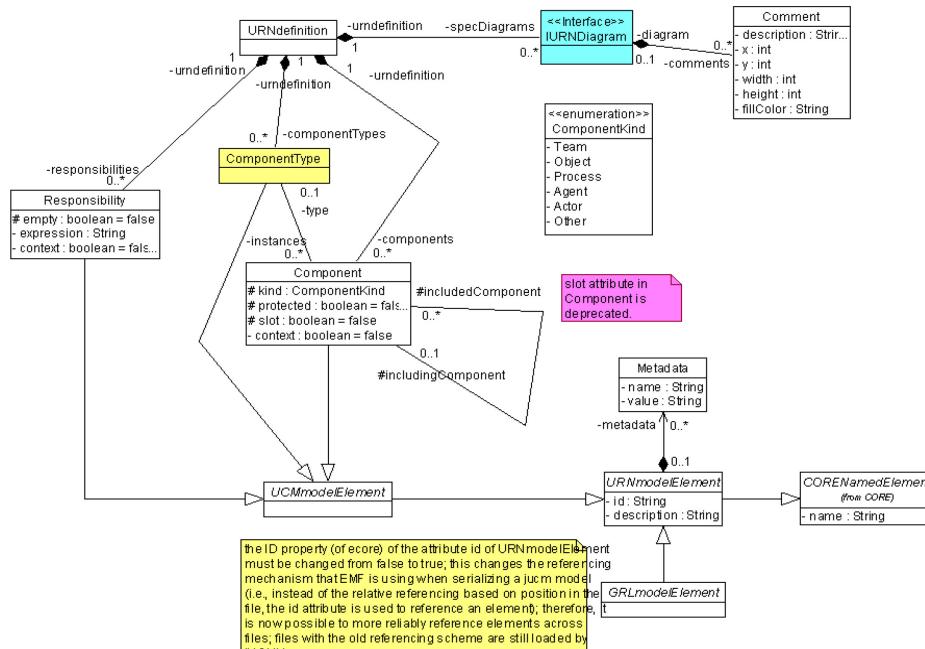


Figure 4: URN Core [38]

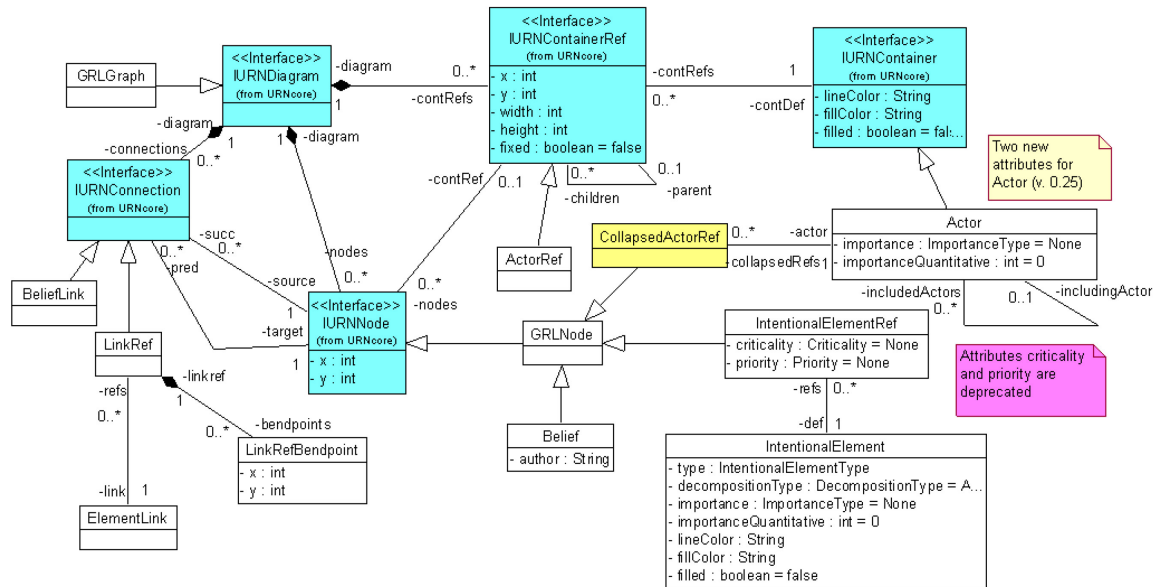


Figure 5: GRL Core [38]

*Actors* and *IntentionalElements* are linkable elements, i.e. they can be linked using *ElementLinks* which can be of three types – *Dependency*, *Contribution* or *Decomposition* (see Figure 6).

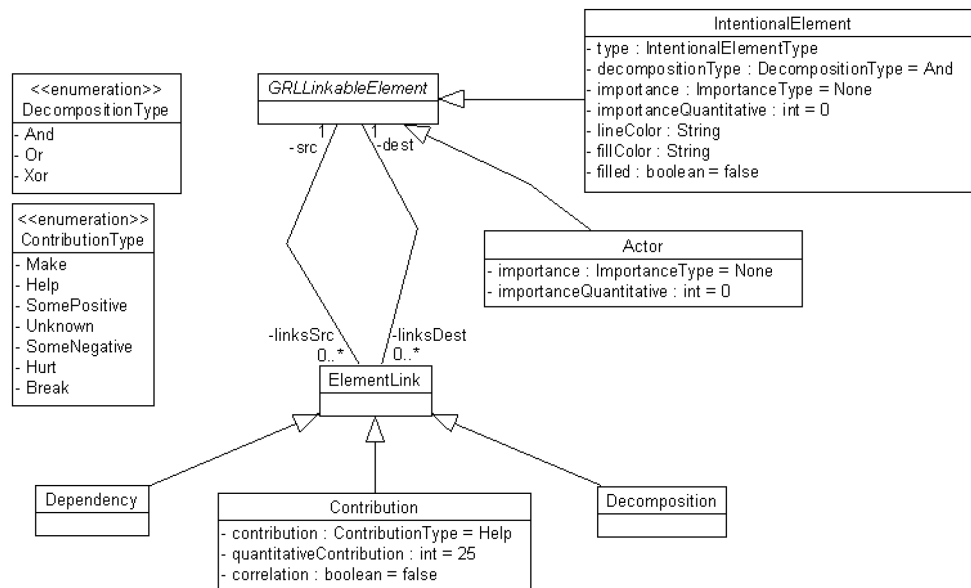


Figure 6: GRL Links [38]

A GRLspec also consists of *StrategiesGroup* used to organize evaluation strategies and manipulate them as a group. An *EvaluationStrategy* is a collection of evaluations used to define satisfaction levels for a subset of intentional elements of GRLspec. An *Evaluation* provides initial quantitative and qualitative evaluation values to an intentional element (see Figure 7).

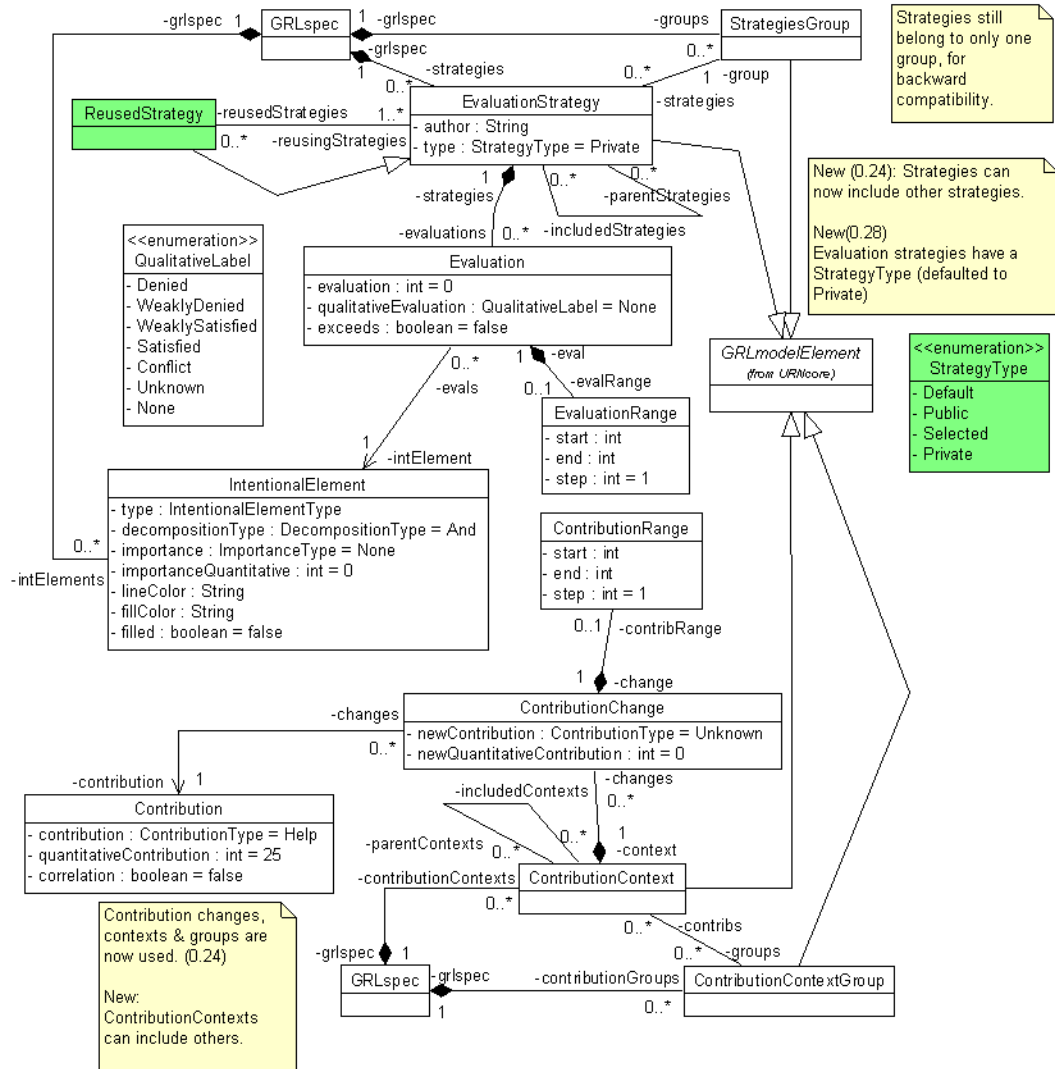
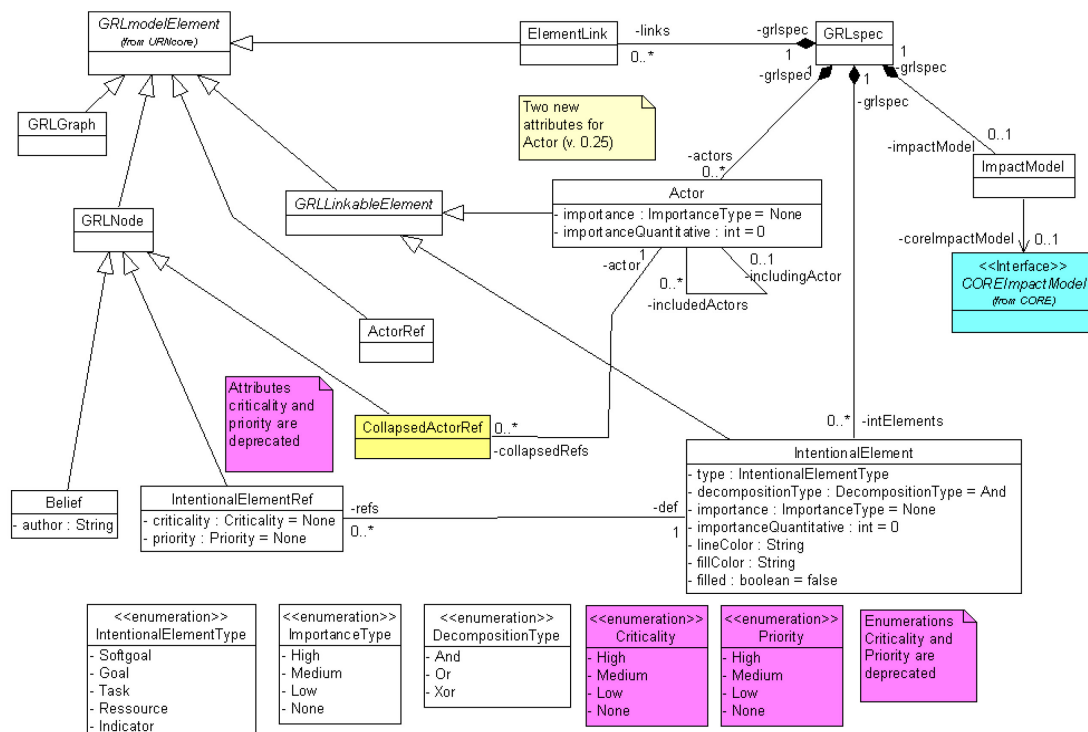


Figure 7: GRL Strategy [38]

Similarly, a *ContributionContextGroup* is a collection of contribution contexts used to organize sets of contribution changes (also known as overrides) and to manipulate them as a

group. A *ContributionContext* is a collection of contribution changes where a *ContributionChange* provides quantitative and qualitative contribution attributes that override an existing contribution (see Figure 7).



In case of UCM, *UCMmap* serves as a container for the UCM specification elements of a single map consisting of PathNodes and ComponentRefs. Responsibilities and Components are contained in the URNdefinition and linked with the corresponding RespRefs and

ComponentRefs, respectively. Similar to GRL, the path nodes have NodeConnections, each of which may also contain a Condition (see Figure 9).

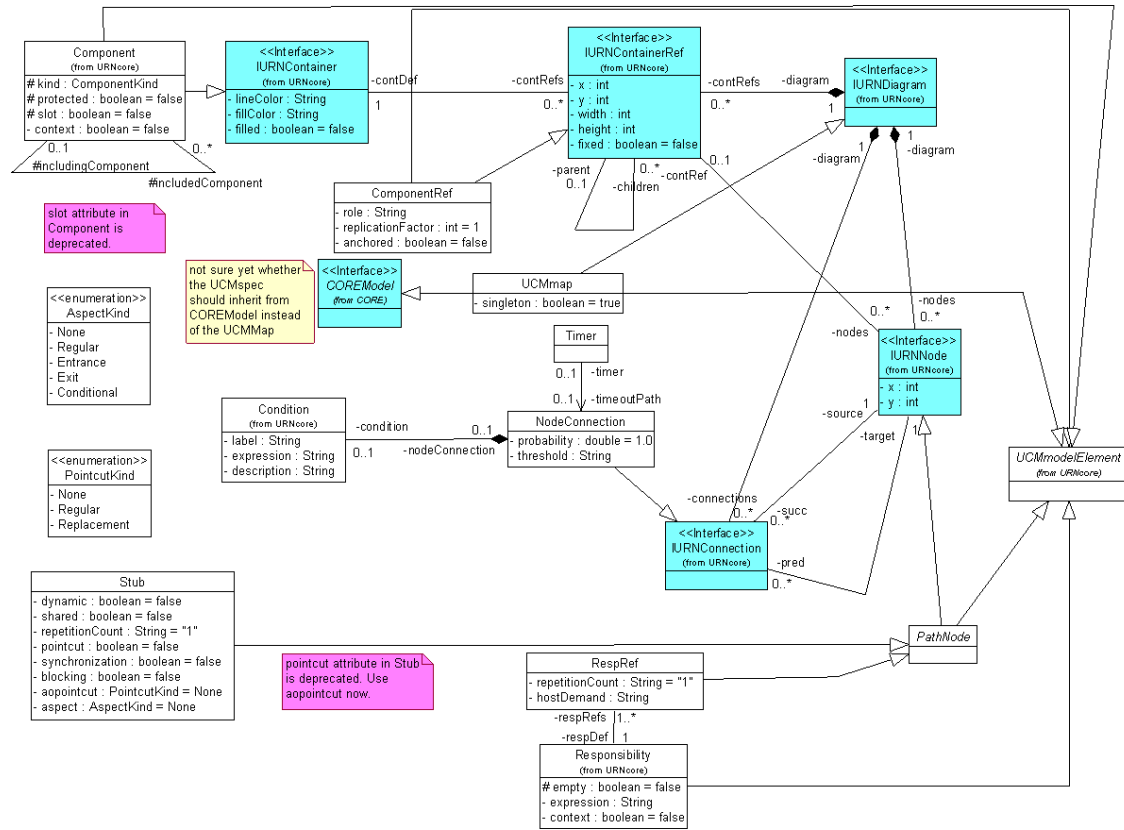


Figure 9: UCM Map Links [38]

A *PathNode* can be of type WaitingPlace/Timer, OrFork, AndFork, OrJoin, AndJoin, RespRef, StartPoint, EndPoint, EmptyPoint, Connect, FailurePoint or a Stub as shown in Figure 10. All of these path nodes have already been explained in Section 2.1.2 except for EmptyPoint, Connect, and FailurePoint. An *EmptyPoint* is a path node that is used for laying out the path of a map and also to asynchronously connect two paths. A *Connect* is used to connect two paths synchronously (i.e., in sequence by connecting an EndPoint to another path) or asynchronously (i.e., in passing by connecting an EmptyPoint to another path). A *FailurePoint* indicates the location of a failure on a scenario path.

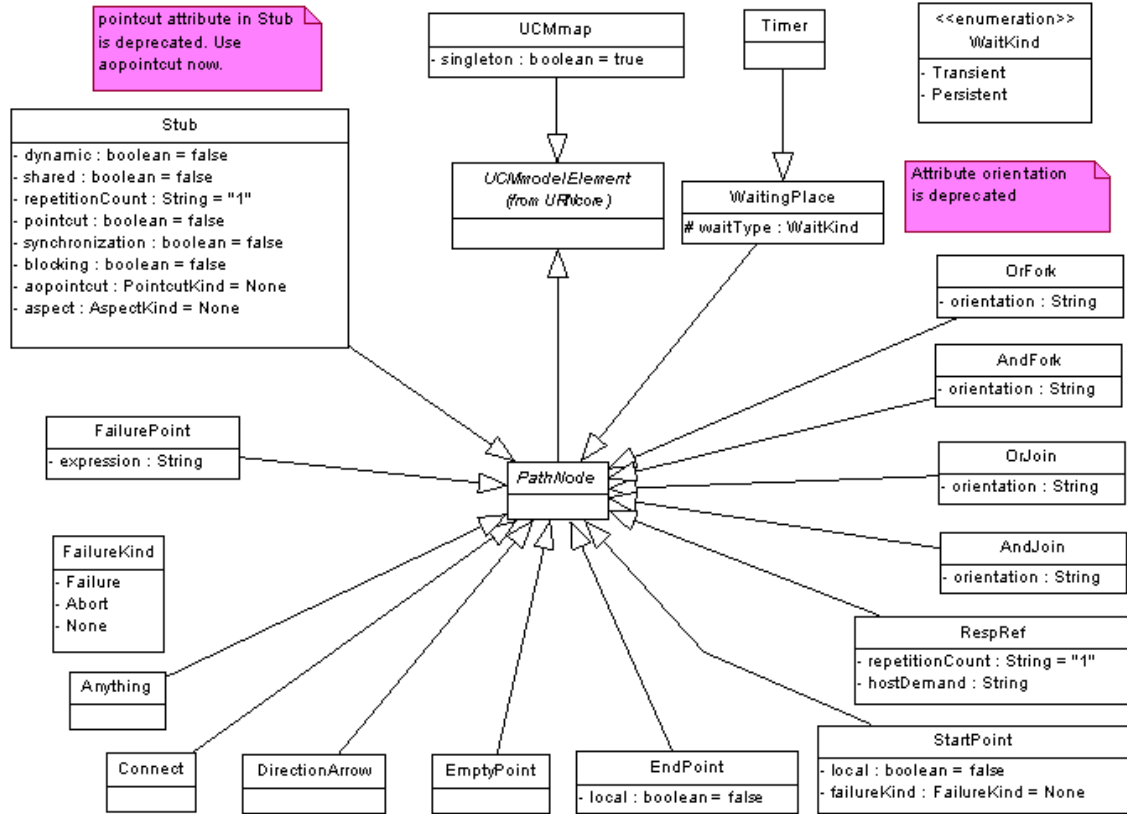


Figure 10: UCM PathNodes [38]

A Stub also consists of *PluginBindings* to determine the continuation from a stub on a parent map to start points and from end points on the stub's plugin map. Each plugin binding has at least one *InBinding* and an *OutBinding*. An *InBinding* connects a *NodeConnection* indicating the stub entry path with a plugin map's *StartPoint*. An *OutBinding* connects a *NodeConnection* indicating the stub exit path with a plugin map's *EndPoint*. *PluginBindings* can also contain responsibility bindings, component bindings and certain conditions for path traversal (see Figure 11), which are out of scope for this work.

The UCMspec also contains various UCM elements for the description of scenario definitions as well as performance annotations, which are out of scope for this work and hence not further discussed.

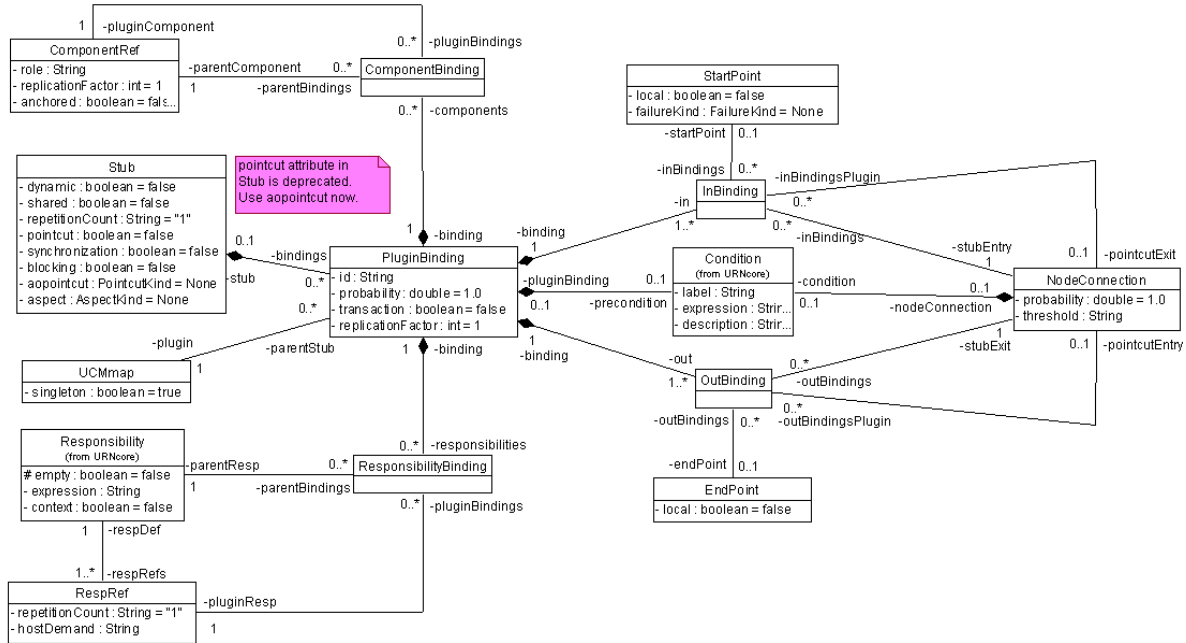


Figure 11: UCM PluginBindings [38]

Last but not least, a *Concern* is used to group related GRL and UCM diagrams into one unit of understanding. This grouping can be guarded with a *Condition* for composition purposes (see Figure 12).

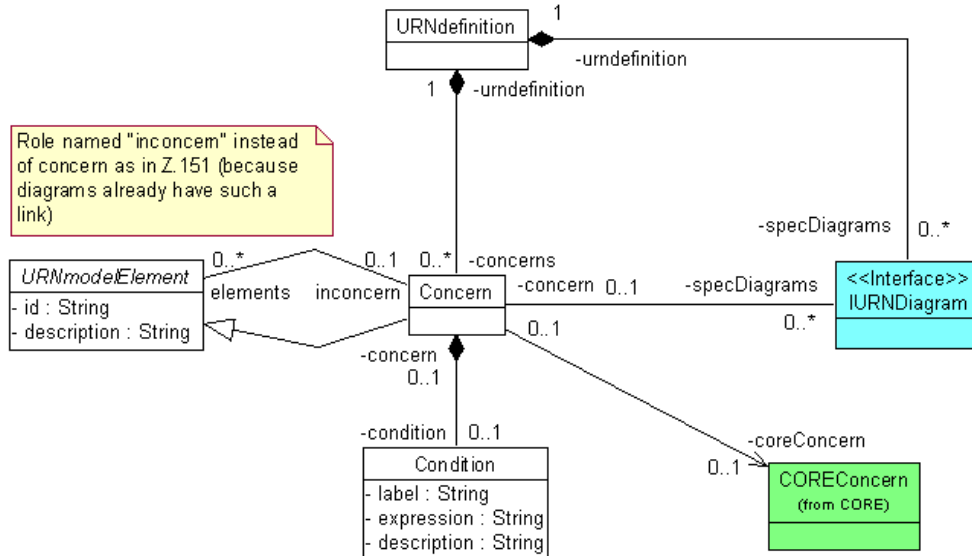


Figure 12: URN Concern [38]

## ***2.3 Summary***

This chapter discusses the essential background information needed about URN before diving into the concepts of Textual URN in the next chapter. The basic GRL and UCM notations are briefly described along with some examples for illustration. Moreover, the URN metamodel is presented to provide the foundation for the TURN metamodel in the next chapter.



## CHAPTER 3: Textual URN Specification

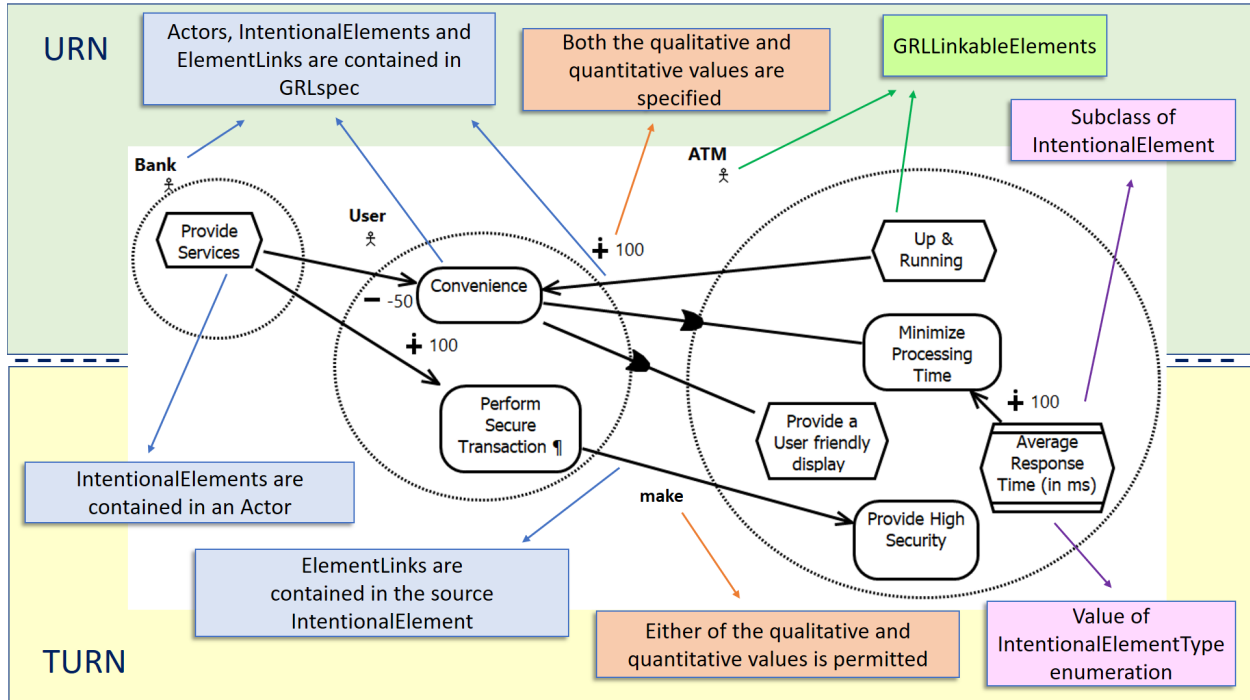
This chapter first highlights the differences between the TURN metamodel and URN metamodel with the help of a concrete URN example. It, then, presents the TURN metamodel with some examples of concrete TURN syntax. Finally, it introduces the Xtext framework including the key features of an Xtext specification language and the proposed TURN grammar.

### *3.1 Differences to URN Metamodel*

The TURN specification strikes a balance between supporting as many language features of URN as possible and the usability, convenience and expediency of the textual syntax. Therefore, not all URN concepts are supported and hence, a model based on a TURN specification does not have the ability to specify everything that the URN model can specify. The differences are highlighted, laying the foundation for a description on how to transform a model conforming to the TURN metamodel into a model conforming to the URN metamodel.

The TGRL (Textual GRL) grammar matches the GRL metamodel to a large degree except that there are some minor differences presented below (also depicted in Figure 13):

1. The inheritance hierarchy of `GRLmodelElement`, `GRLLinkableElement`, and `GRLContainableElement` is flattened, i.e., these classes do not appear in the TURN grammar. However, this does not cause any compatibility issues between TURN and URN, because these classes are abstract and all their attributes and associations are pushed to their subclasses.



**Figure 13: Overview of Main Differences to GRL Metamodel**

2. The Indicator in TURN is merged into IntentionalElement, and the unit attribute of Indicator is now an optional element of IntentionalElement specified only for IntentionalElements of type indicator. Consequently, the IntentionalElementType now includes also indicator.
3. An Actor is not linkable in TURN (i.e., does not inherit from GRLLinkableElement) as showing links among actors is a visualization issue and not as useful for a textual specification.
4. An IntentionalElement, an Indicator, and an ElementLink are contained in the GRLspec in the URN metamodel. In the TURN grammar, however, an IntentionalElement is contained in an Actor and an ElementLink is contained in the source IntentionalElement of the link. This does not allow for elements outside of an

actor, but results in a well-nested textual specification that is easier to comprehend and maintain.

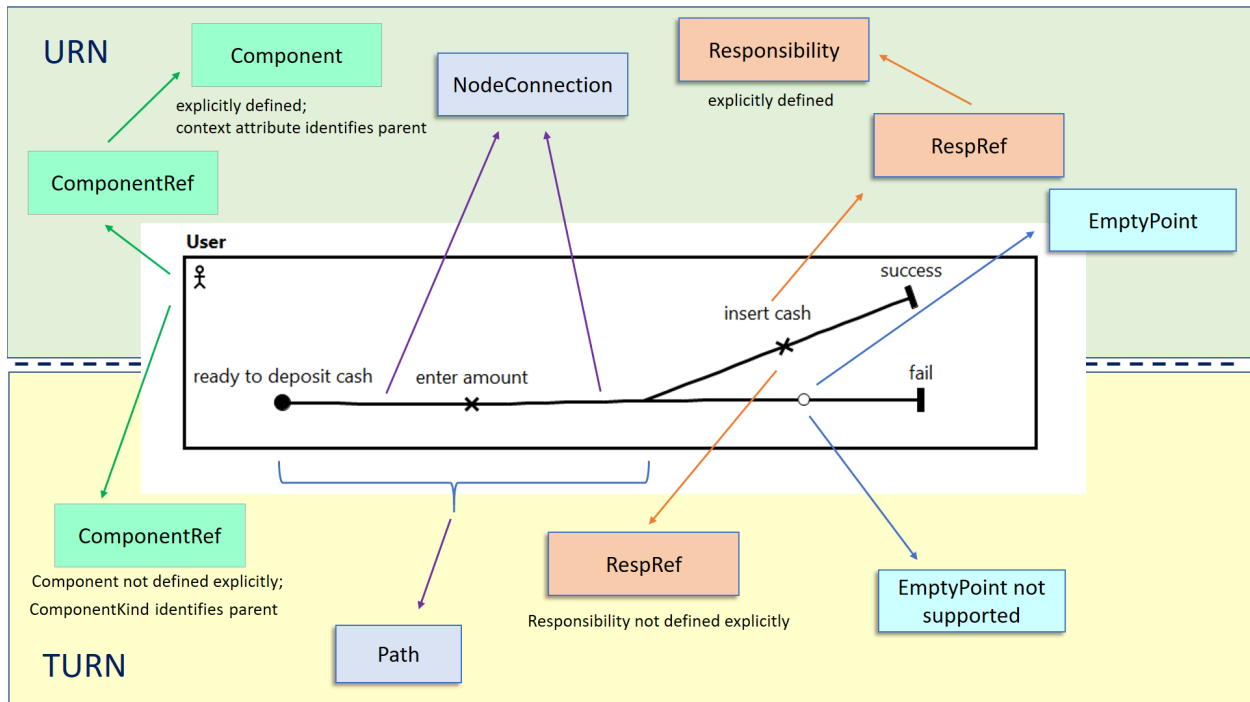
5. The TURN grammar allows either qualitative or quantitative importance/contribution values to be specified for an element but not both at the same time.

The TUCM (Textual UCM) grammar matches the UCM metamodel to some extent, but there are significant differences as depicted in Figure 14. In addition, some other differences exist because a few UCM concepts are not yet supported by TUCM:

1. The most important difference is that the TURN grammar is structured around the notion of Path, which does not exist in the URN metamodel. The abstract class PathNode is flattened, i.e., this class does not appear in the TURN grammar. However, this does not cause any compatibility issues between TURN and URN, because all its attributes and associations are pushed to its subclasses. Furthermore, NodeConnections are specified implicitly through the order of path nodes in the TURN specification. Instead of PathNode and NodeConnection, the TURN grammar uses Path, PathBody, PathWithRegularEnd, PathWithReferencedEnd, PathWithReferencedStub, PathBodyNodes, PathBodyNode, RegularEnd, and ReferencedEnd.
2. Component and Responsibility are specified indirectly with the *name/longName* of ComponentRef and RespRef, respectively.
3. The context attribute of a Component in URN is replaced by the additional value “parent” in the ComponentKind enumeration in TURN.

4. The TURN grammar does not support EmptyPoints and ComponentTypes.
5. The TURN grammar does not support UCM performance specifications including the probability attribute of NodeConnection and PluginBinding.
6. The TURN grammar does not support scenario definitions (ScenarioDef) including ScenarioGroups and Variables.

Empty points are irrelevant for a textual syntax as they are mostly required for graphical layouting. Component types and performance annotations are rarely used and scenario definitions are left for future work.

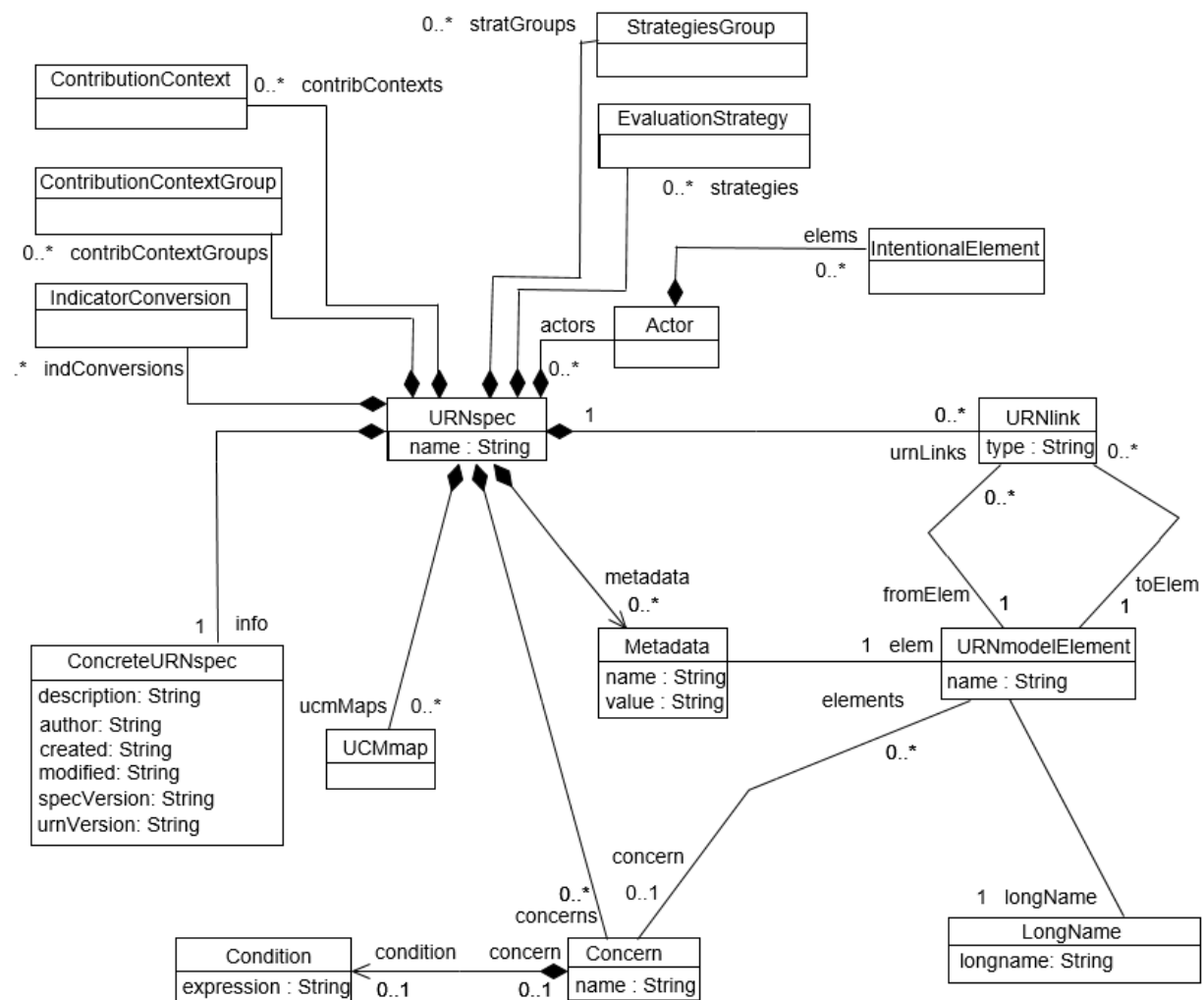


**Figure 14: Overview of Main Differences to UCM Metamodel**

Figure 14 only gives an overview of the main differences to the UCM Metamodel and other more detailed differences are explained in the following Section 3.2.

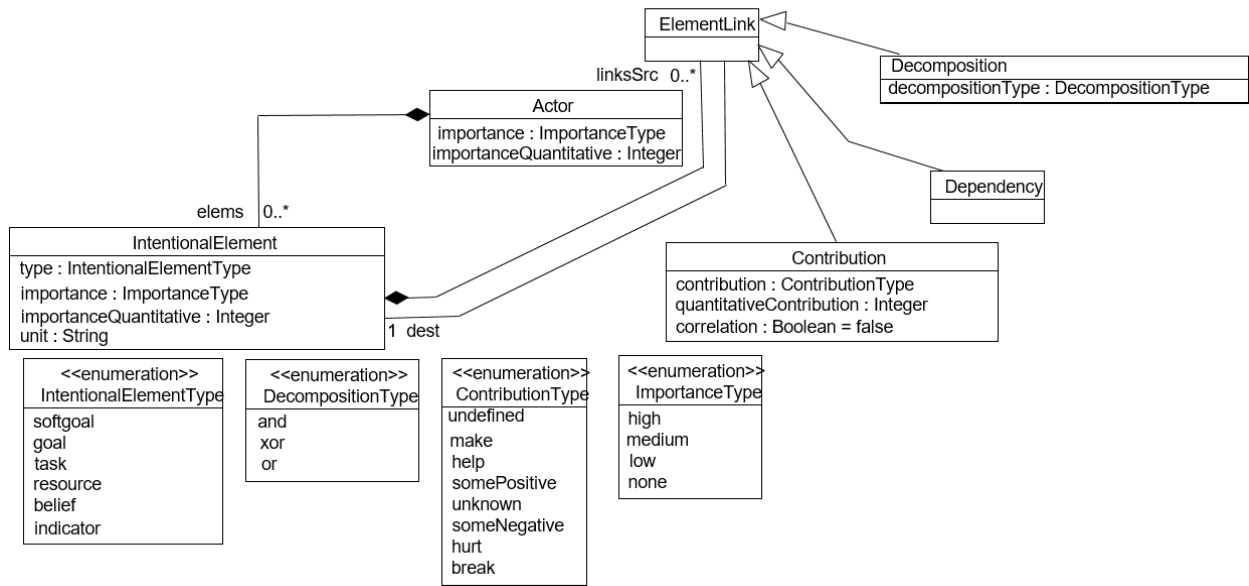
### 3.2 *TURN Metamodel*

Similar to the URN metamodel, the TURN metamodel also has a root container called URNspec that consists of a separate class ConcreteURNspec for representing detailed information. It also consists of actors, urnLinks, concerns, ucmMaps, indConversions (IndicatorConversion), contribContextGroups (ContributionContextGroup), contribContexts (ContributionContext), componentRefs, stratGroups (StrategyGroup), strategies and metadata for all URNmodelElements (see Figure 15). As in URN, the URNmodelElement is the superclass for all URN model elements (not shown in the figures to avoid cluttering).



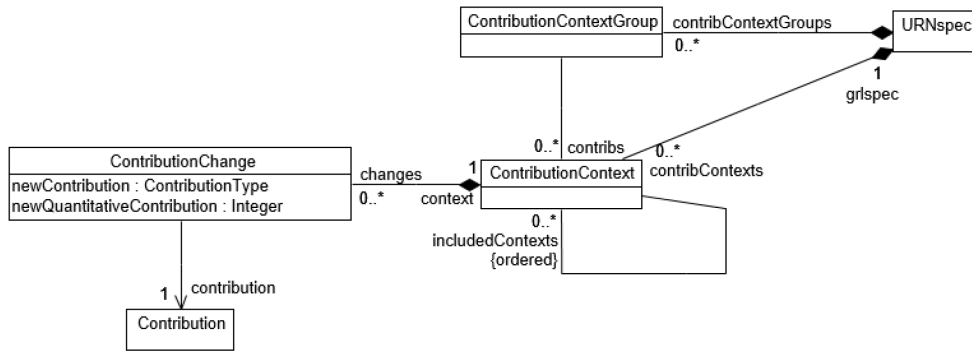
**Figure 15: TURN Main**

As discussed in Section 3.1, the containment hierarchy is different for TURN compared to URN: an Actor contains IntentionalElements and an IntentionalElement further contains ElementLinks. This containment hierarchy is shown in Figure 16 where an ElementLink can be of type Contribution, Dependency or Decomposition. This difference makes it easier to specify a textual model, because the TGRl specification is well-nested.



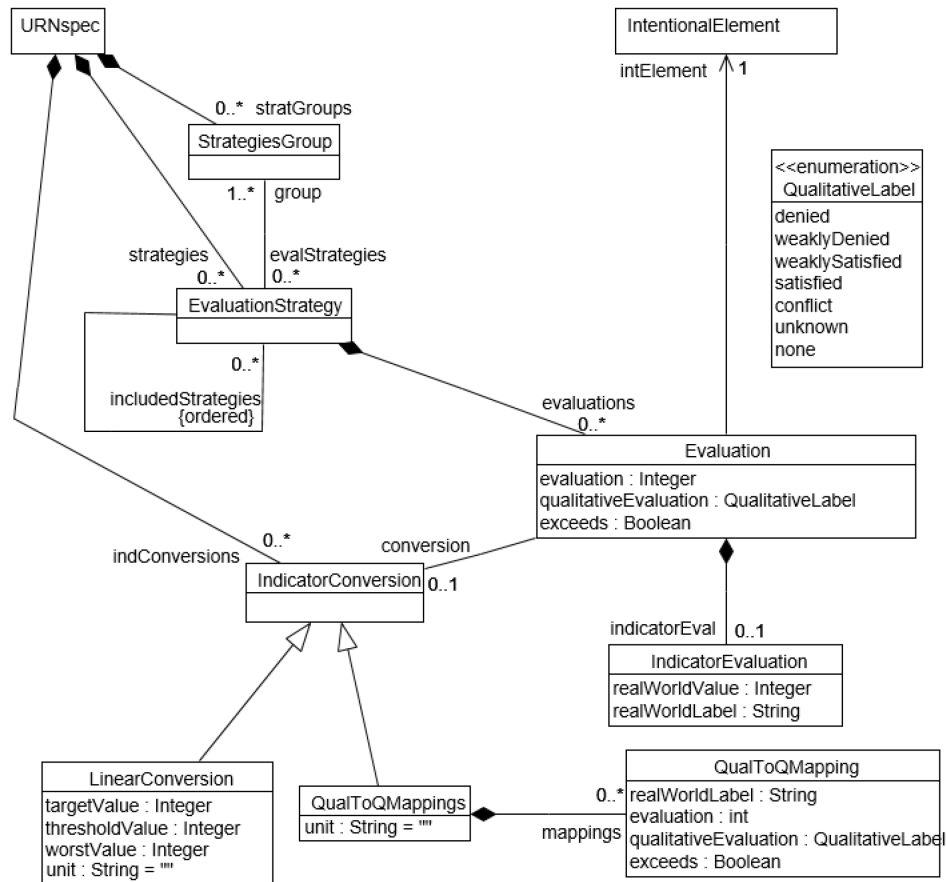
**Figure 16: TGRl Intentional Elements and Element Links**

A ContributionContextGroup is connected to ContributionContexts that can further include ContributionChanges and other ContributionContexts where a ContributionChange must be connected to a Contribution. This is similar to GRL metamodel except that the TGRl metamodel allows either new qualitative or new quantitative contribution values to be specified for a ContributionChange but not both at the same time. Furthermore, contexts and groups are contained in GRLspec in URN, whereas they are contained in the URNspec in TURN.



**Figure 17: TGRl Contribution Context**

The StrategyGroups and Evaluations in TURN match the ones in URN (see Figure 18) except that the URNspec contains strategies, groups and conversions instead of the GRLspec.



**Figure 18: TGRl Strategies and Evaluation**

For UCM, the differences are more pronounced (see Figure 19 and Figure 20). Whereas NodeConnections connect source and target path nodes with each other in URN, the concept of Path is used for TURN. A Path represents a segment of a UCM map. Segments correspond to the PathBodyNodes of a UCM map that lie between branching points (e.g., forks, joins and stubs). A branch is typically treated as a separate path in TURN. This is done because a textual representation requires a tree-based format, and hence the graph-based representation of a UCM map is broken down into individually specifiable segments.

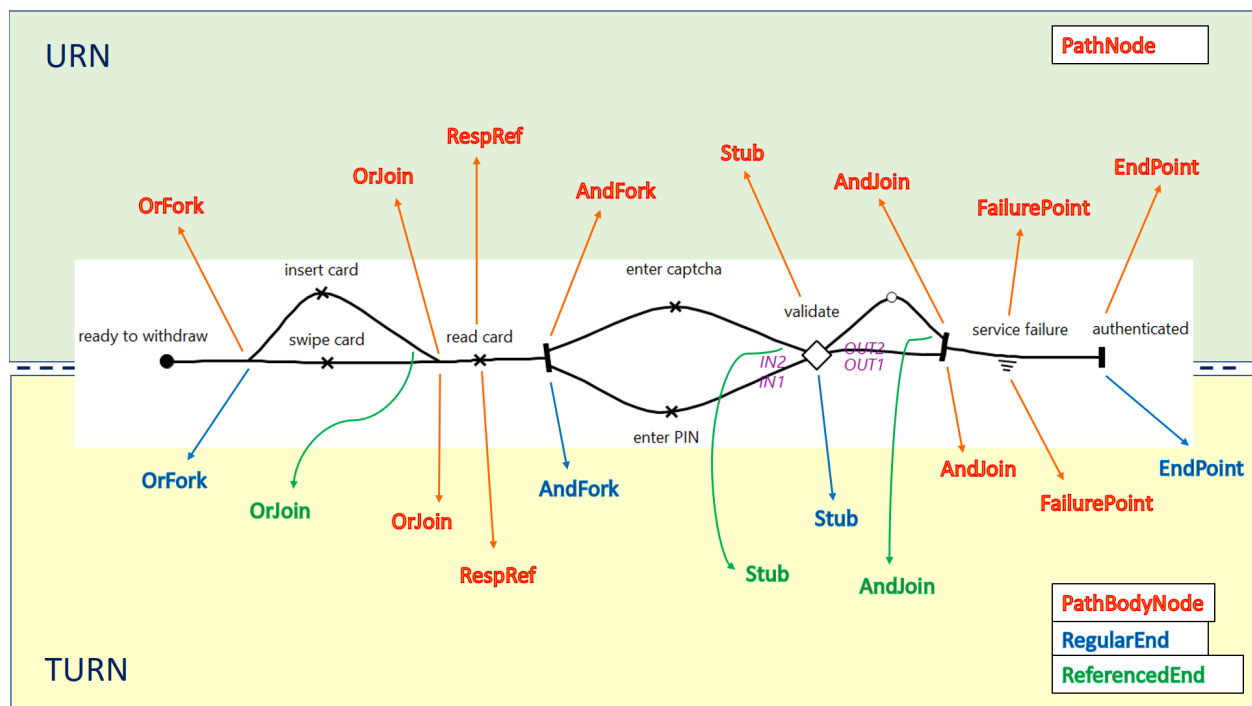


Figure 19: Key Difference Between UCM and TUCM – Paths

For example, the UCM in Figure 19 contains the following seven Paths when modeled with TURN: (i) from StartPoint to OrFork, (ii) from OrFork to AndFork via “swipe card”, (iii) from OrFork to OrJoin via “insert card”, (iv) from AndFork to Stub via “enter PIN”, (v) from AndFork to Stub via “enter captcha”, (vi) from Stub to EndPoint via out-path 1, and (vii) from Stub to AndJoin via out-path 2. The last node in a Path may either be a *RegularEnd*, i.e., the



actual definition of a PathBodyNode (OrFork, AndFork, Stub and EndPoint in Figure 19), or a *ReferencedEnd*, i.e., a reference to an already specified PathBodyNode (OrJoin, AndJoin and Stub in Figure 19).

Furthermore, TURN does not require OrJoins to be specified that are implied by the structure of the UCM model (see Figure 20). For example, a responsibility (RespRef), FailurePoint, or EndPoint may be referenced directly on a Path without having to explicitly specify a preceding OrJoin as is required in a valid URN model. Instead, the transformation from TURN to URN ensures that the representation of a TUCM model with implicit OrJoins shown at the bottom of the figure is properly translated into a valid UCM model with explicit OrJoins shown at the top of the figure.

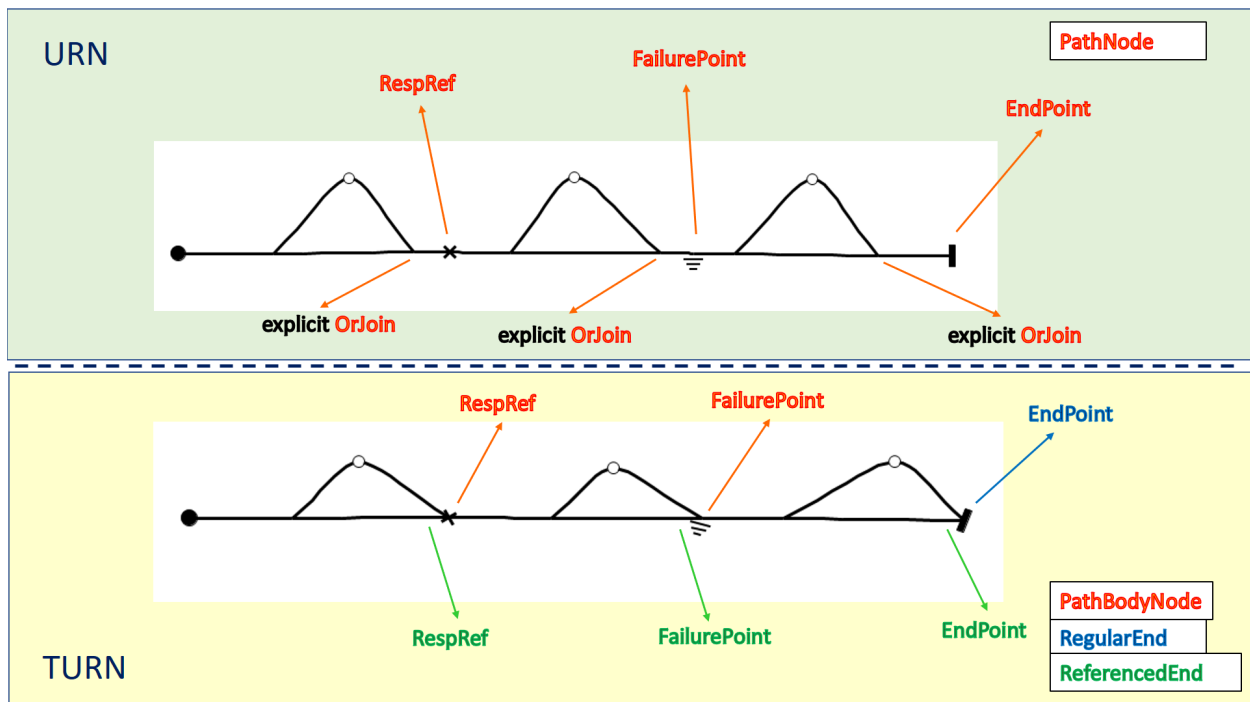
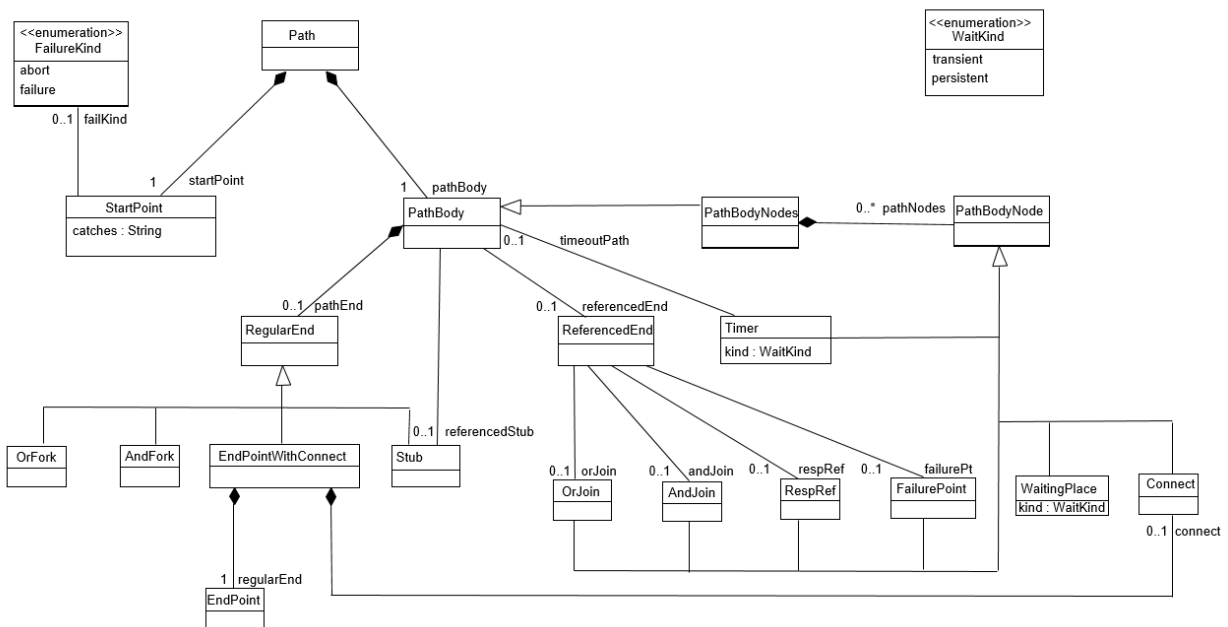


Figure 20: Key Difference Between UCM and TUCM – Implicit OrJoins

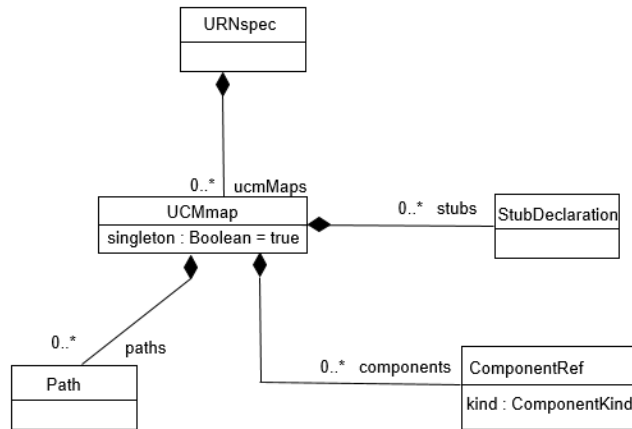
The concept of a Path and all other concepts mentioned in this paragraph except for standard URN path nodes do not exist in URN. A *Path* contains a StartPoint and a PathBody

where a *PathBody* contains a set of *PathBodyNodes*, which cover all supported URN path nodes. In addition, a *PathBody* can have a *RegularEnd*, a *ReferencedEnd* or a *ReferencedStub* as shown in Figure 21. A *RegularEnd* denotes the end of a path segment, which can either be an *EndPoint* or a node where new branches start (i.e., an *OrFork*, *AndFork* or *Stub*). A *ReferencedEnd* and *ReferencedStub* denote the end of path that ends at a path node defined in a different path (*OrJoin*, *AndJoin*, *RespRef*, *FailurePoint* and *Stub*, respectively).



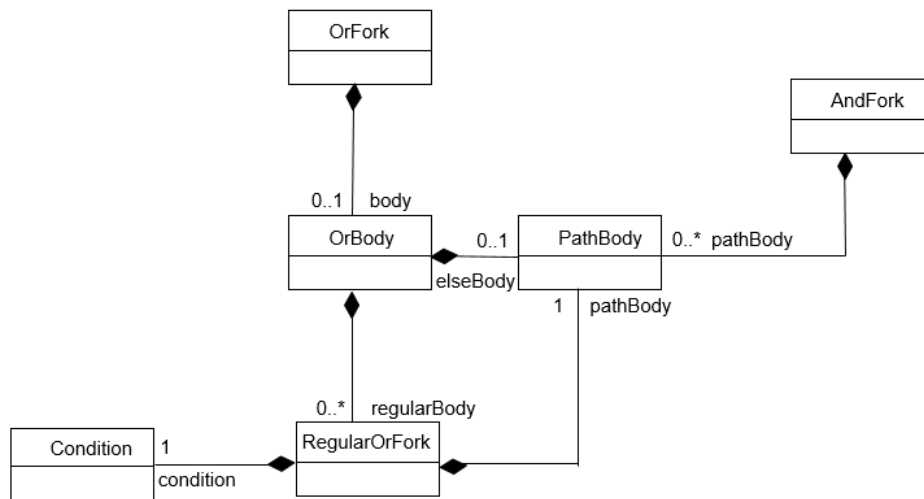
**Figure 21: TUCM Path**

The TURN UCMmap consists of Paths and ComponentRefs (see Figure 22). A *ComponentRef* is equivalent to a ComponentRef in URN with its name/longname implicitly representing a Component in URN. Furthermore, *StubDeclarations* may be defined to allow details of the plugin bindings of a stub to be defined outside the main path specification, hence decluttering the textual path specification.



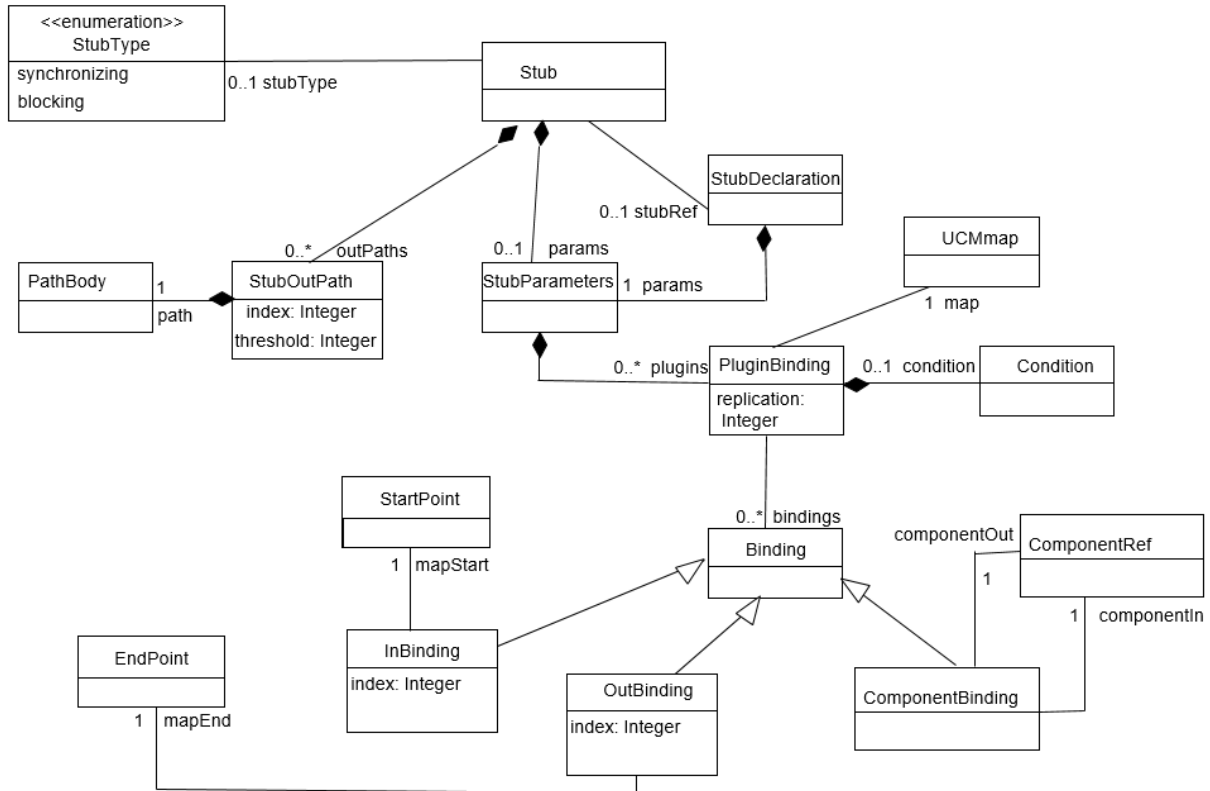
**Figure 22: TUCM Map**

An *OrFork* in TURN also has a different hierarchical structure as compared to an *OrFork* in URN. It contains an *OrBody* which may further contain *RegularOrForks*, i.e., a group of a condition and a pathBody. The *OrBody* also may have an *elseBody* (pathBody) to be followed when all other conditions are false. An explicit else branch does not exist in the URN metamodel, but is supported by a *NodeConnection* with a condition set to “else”. An *AndFork* contains multiple pathBodies.



**Figure 23: TUCM OrFork and AndFork**

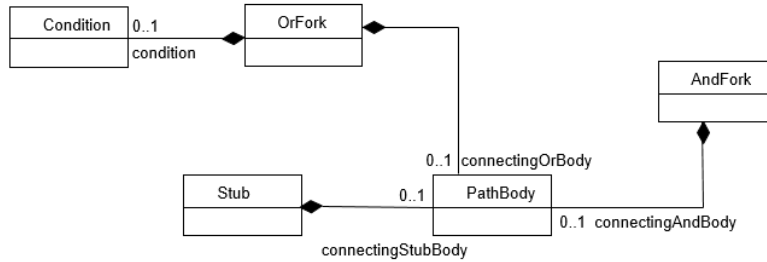
A *Stub* in TURN is structurally a bit different from the one in URN. It may have a reference to a *StubDeclaration* in order to allow the specification of *PluginBindings* outside the path, or may contain *StubParameters* that directly carry *PluginBindings*. A *Stub* can have several in-paths identified by unique indexes. A *Stub* also has *StubOutPaths* each of which is identified by an index and contains a *PathBody*. Each of the *PluginBinding* is connected to a *UCMmap* (i.e., the plugin map) and *Bindings* like *InBinding* and *OutBinding*. An *InBinding* is connected to a *StartPoint* and an indexed in-path, while an *OutBinding* is connected to an *EndPoint* and an indexed outpath. A *PluginBinding* may also contain a condition for navigation to the plugin map.



**Figure 24: TUCM Stub and Bindings**

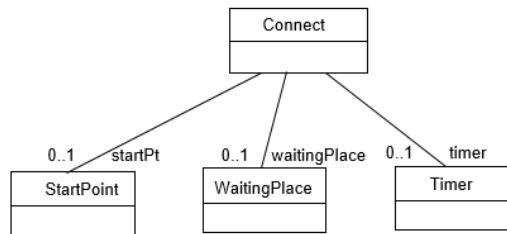
PathNodes like *OrFork*, *AndFork* and *Stub* may also have a *connecting path body* to represent the continuation of their branches in the case of *OrFork* and *AndFork* and outpaths in the case of a *Stub* (see Figure 25). The connecting path body represents an implicit *OrJoin*

(AndJoin in the case of an AndFork), i.e., the branches/outpaths first join before continuing with the connecting path body. A connecting path body is a shortcut notation to avoid having to specify shared path nodes in each branch/outpath or having to specify explicit joins. One example is given in map TL where the path continues to responsibility deny after an OrFork (indicated with two “;”, one at the end of the branch of the OrFork and one at the beginning of the connecting path body – see Listing 4). Another shortcut notation is an OrFork that just consists of a Condition. This is useful for OrForks with two branches where one branch does not have any path nodes and hence does not need to be shown.



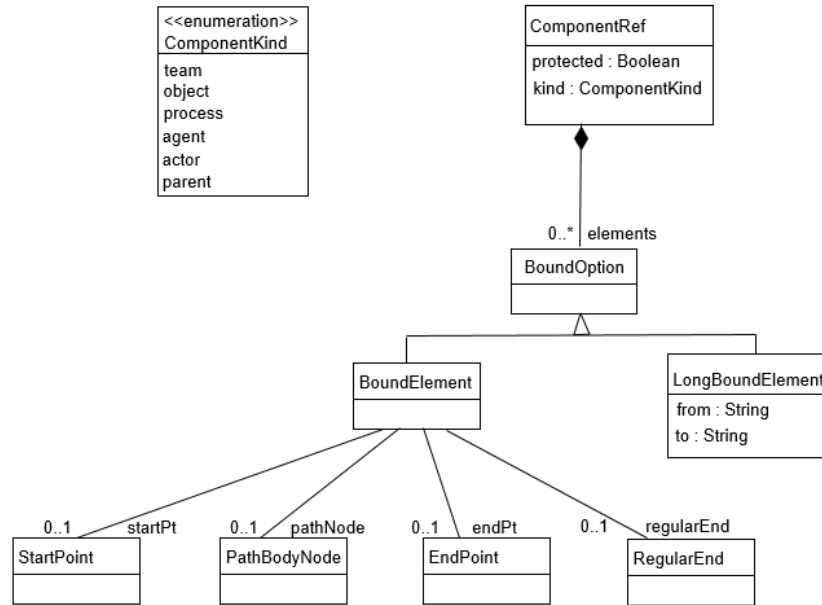
**Figure 25: Connecting Path Bodies in OrFork, AndFork and Stub**

Just as in URN, there are three types of *Connects* in TURN – StartPoint, WaitingPlace and Timer as shown in Figure 26. A Connect represents either the synchronous Connect from URN between an explicit EndPoint of one Path and a StartPoint, WaitingPlace or Timer on another Path or the asynchronous Connect from URN between an implicit empty point on one Path and a StartPoint, WaitingPlace or Timer on another Path.



**Figure 26: TUCM Connects**

The binding of elements to a *ComponentRef* can be done in two ways – either by referencing the elements directly (*BoundElement* in Figure 27) or by defining a starting element and an end element to bind all the elements that occur in between including them (*from* and *to* in *LongBoundElement*, respectively, in Figure 27). URN only supports the first option.



**Figure 27: TUCM ComponentRefs**

### 3.3 Examples of Concrete Syntax

This section shows the concrete syntax of TURN for the graphical URN examples in Figure 1 and Figure 2. The examples refer to the corresponding classes of the TURN metamodel introduced in Section 3.2 as well as the TURN grammar in Appendix A. An example TURN specification corresponding to the URNspec (urnModel), ConcreteURNspec, Concern, URNlink, and Metadata classes in the TURN metamodel and the same rules in the TURN grammar is given below in Listing 1.

**Listing 1: Example TURN for URN Top Level**

```
urnModel Example {  
    description "This is an example."  
    author "Ruchika Kumar"  
    created "Oct-15-2016"  
    modified "Oct-15-2016"  
    version "1.0"  
    urnVersion "29.0"  
}  
  
concern TCConcern : TelP, TCS, TL  
  
link TelP --> OriginatingAgent  
link TelP --> TerminatingAgent  
  
metadata TCS: popularity = 90  
metadata TL: popularity = 35  
}
```

An example TURN specification corresponding to the Actor and IntentionalElement classes including all types of IntentionalElements in the TURN metamodel and the same rules in the TURN grammar is given below in Listing 2. The below TURN specification also features Contributions (contributesTo), Decompositions (decomposes), and Dependencies (dependsOn).

**Listing 2: Example TURN for GRL Core**

```
actor TelP#"Telecom Provider" {  
    importance 100  
    goal VoiceConn#"Voice Connection Be Setup" {  
        importance 50  
    }  
  
    softgoal HighRel#"High Reliability" {  
        importance 75  
    }  
  
    softgoal SpecUsage#"Minimize Spectrum Usage" {  
        importance 60  
    }  
  
    task MakeVoiceOverInternet#"Make Voice Connection Over Internet" {  
        contributesTo HighRel with somePositive  
        contributesTo SpecUsage correlated with somePositive  
        xor decomposes VoiceConn  
    }  
}
```

```

task MakeVoiceOverWireless#"Make Voice Connection Over Wireless" {
    contWirelessVoiceConnToHighRel contributesTo HighRel with make
    contributesTo SpecUsage correlated with someNegative
xor decomposes VoiceConn
}

indicator VoiceConnFailureRate#"Failure Rate for Voice Connection Over
Internet" {
    unit "failures/week/10000 connections"
    contVoiceConnFailureRateToInternetVoiceConn contributesTo
    MakeVoiceOverInternet with 100
    dependsOn Tech.LoggEquip
}

belief WirelessReliability#"Wireless is less reliable than Internet" {
    contributesTo HighRel with someNegative
}

actor Tech#"Technician" {
    resource LoggEquip#"Logging Equipment" {
        dependsOn EquipSetup
    }

    task EquipSetup#"Correctly setup logging equipment" {
        importance 100
    }
}

```

The concrete syntax for the top level URN classes (see Listing 1) and TGRL (see Listing 2 and Listing 3) follows a typical approach for textual languages, quite similar to OMG's Human-Usable Textual Notation (HUTN) standard [33]. A model concept is represented by a unique keyword (e.g., actor, goal, or dependsOn) and additional information about the concept is shown in curly brackets (e.g., actor { ... }). An attribute of a concept is also identified by a unique keyword and followed by its specific values (e.g., importance 100). This general approach is used for the complete Textual GRL.

As a further example, the TURN specification corresponding to the StrategiesGroup, EvaluationStrategy (strategy), Evaluation, LinearConversion, QualToQMappings



(mappingConversion), ContributionContextGroup, ContributionContext, and ContributionChange classes in the TURN metamodel and the same rules in the TURN grammar is given below in Listing 3.

**Listing 3: Example TURN for GRL Strategies and Contribution Changes**

```

strategiesGroup SG1: Eval1

strategy Eval1#"Internet Connection" {
  author "ITU-T"
  Tech.EquipSetup evaluation 100
  TelP.VoiceConnFailureRate real 265 convertedWith LC1
}

linearConversion LC1#"Weakly Failures" {
  unit "failures/week/10000 connections"
  target 0
  threshold 500
  worst 10000
}

mappingConversion MC1#"EquipmentClassification" {
  unit "equipment class"
  real "Class 1" --> 100
  real "Class 2" --> 47
  real "Class 3" --> -25
}

contributionContextGroup CCG1: CCP, CCPI

contributionContext CCP#"Pessimistic" {
  contWirelessVoiceConnToHighRel with help
}

contributionContext CCPI#"PessimisticIneffective" {
  contVoiceConnFailureRateToInternetVoiceConn with 50
  includes CCP
}

```

The TURN specification in Listing 4 is an example that corresponds to most UCM classes in the TURN metamodel and the same rules in the TURN grammar. The listing shows UCMmaps (map), StartPoints (start), EndPoints (end), RespRefs (X), Stubs (stub) with inpaths (in) and outpaths (out), OrForks (or), AndForks (and), Timers (timer), and ComponentRefs (team and parent).

Listing 4: Example TURN for UCM

```

map SC#"Simple Connection" {
  start request -> stub Originating(
    OF: success=out 1, failPoint=out 2, startPoint=in 1,
    Agent=OriginatingAgent
  ) {
    out 1 -> stub Terminating(
      TF: success=out 1, reportSuccess=out 2, busy=out 3,
      failPoint=out 4, startPoint=in 1, Agent=TerminatingAgent
    ) {
      out 1 -> end ring.
      out 2 -> X forwardSignal -> end ringing.
      out 3 -> X forwardSignal -> end busy.
      out 4 -> end notify.
    }
    out 2 -> end notify.
  }
  actor OriginatingUser: request, notify, busy, ringing
  team OriginatingAgent: Originating, forwardSignal
  team TerminatingAgent: Terminating
  team TerminatingUser: ring
}

map OF #"Originating Features" {
  start startPoint -> stub OrigFeatures(
    [!subTL] Default: startPoint=in 1, continue=out 1, Agent=Agent
    [subTL] TL: startPoint=in 1, success=out 1, failPoint=out 2,
    Agent=Agent
  ) {
    out 1 -> X sendRequest -> end success.
    out 2 -> end failPoint.
  }
  parent Agent: startPoint..success, failPoint
}

map TL {
  start startPoint -> X checkTime -> or {
    [!TLactive] -> end success.
    [TLactive] -> timer getPIN {-> deny;} -> or {
      [PINvalid] -> end success.
      [!PINvalid] -> ;
    } -> X deny -> end failPoint.
  }
  start enterPIN -> end e. -> trigger getPIN ;
  parent Agent: startPoint..failPoint, success
  actor OriginatingUser: enterPIN
}

map Default {
  start startPoint -> end continue.
  parent Agent: startPoint, continue
}

```

```

map TF #"Terminating Features" {
  start startPoint -> stub TermFeatures(
    [!subTCS] Default: startPoint=in 1, continue=out 1, Agent=Agent
    [subTCS] TCS: startPoint=in 1, success=out 1, failPoint=out 2,
    Agent=Agent
  ) {
    out 1 -> or {
      [!busy] -> and {
        * -> X ringTreatment -> end success.
        * -> X ringingTreatment -> end reportSuccess.
      }
      [busy] -> X busyTreatment -> end busy.
    }
    out 2 -> end failPoint.
  }
  parent Agent: startPoint..success, startPoint..reportSuccess,
  startPoint..busy, failPoint
}

map TCS #"Terminating Call Screening (TCS)" {
  start startPoint -> X checkTCS -> or {
    [!onTCSlist] -> end success.
    [onTCSlist] -> end failPoint.
  }
  parent Agent: startPoint..success, failPoint, TCSCreeningList
  team TCSCreeningList: checkTCS
}

```

TUCM uses a different approach than TGRL to avoid an overly verbose scenario specification. Instead, the textual notation uses the notion of a Path with PathBodyNodes separated by a -> (e.g., “**start** startPoint -> **end** continue.” defines the simple Default map from Figure 2 with one StartPoint and one EndPoint). A PathBodyNode where a path splits into several branches is described by nesting its branches inside the PathBodyNode as shown below.

```

start startPoint -> X checkTCS -> or {
  [!onTCSlist] -> end success.
  [onTCSlist] -> end failPoint.
}

```

The above TURN specification defines the Terminating Call Screening map from Figure 2 with one StartPoint, one responsibility (RespRef), one OrFork and its two branches with one

Condition and EndPoint each. A PathBodyNode where several paths join is described by one path defining the join node and the other path referencing this join node as shown below.

```
start startPoint -> X checkTCS -> or {
  [!onTCSlist && premiumService] -> X logCall -> join j -> end success.
  [!onTCSlist && !premiumService] -> j;
  [onTCSlist] -> end failPoint.
}
```

In this example, the first branch defines the OrJoin j, while the second branch references the OrJoin j as indicated by the semi-colon. This approach is used for all PathBodyNodes with branches (i.e., OrForks, AndForks and Stubs) and all PathBodyNodes that join branches (i.e., OrJoins, AndJoins and Stubs). In addition, RespRef, FailurePoints and EndPoints may also be referenced as a shortcut notation as explained in Figure 20.

Another shortcut notation allows PathBodyNodes with branches to define the continuation of two or more branches following an Orfork, AndFork or Stub (i.e., a *connecting path body*; see Figure 25). This situation is depicted by a branch continuing after the closing curly brackets of the OrFork, AndFork, or Stub with a semi-colon as shown below (another example is given in map TL in Listing 4).

```
start startPoint -> X checkTCS -> or {
  [!onTCSlist && premiumService] -> X logCall -> ;
  [!onTCSlist && !premiumService] -> ;
  [onTCSlist] -> end failPoint.
} -> end success.
```

The branches that end with a semi-colon are joined by an implicit OrJoin (AndJoin in case of an AndFork) before the connecting path body (i.e., “-> end success.”). For a Stub, out-paths that end with a semi-colon are joined implicitly before the connecting path body.

### ***3.4 Introduction to Xtext Framework & TURN Grammar***

#### **3.4.1 Xtext Framework**

Xtext [40] is a framework intended for the development of DSLs (Domain Specific Languages). It is being developed as part of the Eclipse Modeling Framework (EMF) Project [12]. The classes referred to in this subsection that start with a capital E are from EMF. Xtext is a powerful tool that lets us define the grammar of our language and automatically generate editors for such language. The grammar is written with a set of rules that can be of the following types:

*Terminal Rules:* These are also referred to as *token rules* or *lexer rules*. Definitions of terminal rules represent the lexical analysis of the parser, which is usually realized by regular expressions. A *Terminal Rule* produces a single atomic terminal token and, by convention, is written in upper-case after the word “terminal”. TURN does not require terminal rules to be specified as it uses the basic terminal rules already provided by the Xtext framework (e.g., see *INT* in Listing 5).

*Parser Rules:* Contrary to terminal rules, these rules produce a tree of non-terminal and terminal tokens and are also called *production rules*. These fulfill the role of syntax analysis of the parser. Since the grammar contains an initial rule to specify the root of the Abstract Syntax Tree (AST) [1], it denotes this rule by first position among all the rules. These rules produce instances of EClasses (e.g., see *Actor* in Listing 7).

*Data Type Rules:* These are types of parser rules that produce instances of EDataTypes instead of EClasses. These rules neither contain any actions nor assignments, and the data type

EString is implied if none has been explicitly declared. The language developer is responsible for converting the string to a data type using value converters. One example of a data type rule is shown in Listing 5 and the corresponding value converter rule is shown in Listing 6.

**Listing 5: Example Data type rule**

```
PositiveInteger returns ecore::EInt: INT;
```

**Listing 6: Example Value Converter for Positive Integer**

```
private IValueConverter<Integer> positiveValueConverter = new
IValueConverter<Integer>() {

    @Override
    public Integer toValue(String string, INode node) throws
    ValueConverterException {

        if (Strings.isEmpty(string)) {
            throw new ValueConverterException("Cannot be empty", node, null);
        }
        if(Integer.parseInt(string) <= 0){
            throw new ValueConverterException("value cannot be less than 0",
            node, null);
        }
        try {
            return Integer.parseInt(string);
        } catch (NumberFormatException e) {
            throw new ValueConverterException("'" + string + "' is not a valid
            integer", node, e);
        }
    }

    @Override
    public String toString(Integer value) throws ValueConverterException {
        return Integer.toString(value);
    }
};

@ValueConverter(rule = "PositiveInteger")
public IValueConverter<Integer> positiveIntegerLiteral() {
    return positiveValueConverter;
}
```

*Enum Rules:* The Xtext framework offers special rules that return enumeration literals from strings. They can be regarded as shortcuts for data type rules with specific value converters.

The name of the rule and the textual enumeration follow the “enum” terminal (e.g., see *ContributionType* in Listing 7).

Xtext combines BNF specifications of the concrete grammar with the specification of the abstract grammar (i.e., the metamodel) to describe the textual syntax of a language.

The key features of Xtext required to specify the TURN language are shown below:

1. *Class* – Each parser rule in an Xtext specification corresponds to a *Class* in a metamodel. A parser rule may consist of attributes, compositions or references, e.g., *Actor* is a *class* (see Listing 7).
2. *Attribute* – An *Attribute* has a basic type in a class, e.g., *name* is an attribute of class *Actor* (see Listing 7).
3. *Composition* – A *Composition* refers to another parser rule with a containment (see *IntentionalElement* contained in *Actor* in Listing 7).
4. *Reference* – A *Reference* also enables us to refer to a parser rule and differs from *Composition* by square brackets in terms of grammar specification (see the reference to *IntentionalElement* from *Contribution* in Listing 7).
5. *Assignment* – An *Assignment* is a composite element encapsulating a sub-element. It has three variants: simple (=), multivalued (+=) and boolean assignments (?=) (e.g., *ImportanceType* in *Actor*, *IntentionalElement* in *Actor* and *correlation* in *Contribution*, respectively, in Listing 7).

6. *Cardinality* – Xtext borrows the use of ?, + and \* to indicate optional (zero-to-one), one-or-many and zero-to-many relationships, respectively, from BNF specifications (e.g., the ? for the parentheses surrounding the *importance* attribute in *Actor* and the \* after *IntentionalElement* in *Actor*).
7. *Rule Call* – A *Rule Call* is essentially any call to a terminal or parser rule. It can be either an assignment with rule call, which is a simple rule call, or an unassigned rule call, where the element of the parent parser rule must correspond to one of the sub-elements that are separated by a vertical bar, e.g., *LongName* is a simple rule call (see Listing 7) and *ElementLink* is an unassigned rule call (see Listing 8).
8. *Keyword* – A *Keyword* is an arbitrary string of any length enclosed in quotation marks. If there is a keyword in a rule that means the specification needs to have that keyword at the specified position, e.g., ‘actor’ (in blue) in *Actor* (see Listing 7).
9. *Unordered Groups* – An Unordered Group is used to specify elements in any order in the specification. It is created by using a “&” symbol – typically at the end of a line – between the members of the group (see rule *URNspec* in Appendix A).

**Listing 7: Xtext features example**

```
Actor: 'actor' name=QualifiedName
      longName=LongName '{'
      ('importance' (importance=ImportanceType |
                    importanceQuantitative=QuantitativeValue))?
      elems+=IntentionalElement*
      '};

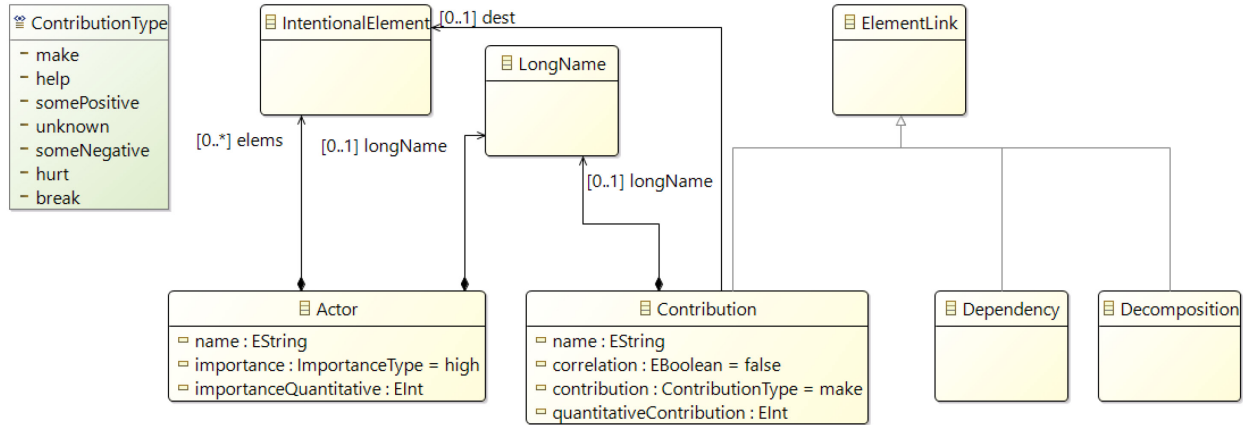
Contribution:
  (name=QualifiedName longName=LongName)?
  'contributesTo' dest=[IntentionalElement | QualifiedName]
  (correlation?='correlated')? 'with' (contribution=ContributionType |
    quantitativeContribution=QuantitativeValue);
```



```
enum ContributionType: make | help | somePositive | unknown | someNegative |
hurt | break;
```

**Listing 8: Unassigned Rule Call example**

```
ElementLink: Contribution | Decomposition | Dependency;
```



**Figure 28: Metamodel corresponding to Xtext features example in Listing 7 and Listing 8**

Figure 28 shows the metamodel that corresponds to the grammar specifications in Listing 7 and Listing 8. The Xtext framework generates this metamodel automatically from the grammar specification. The Actor and Contribution classes with all their attributes correspond to the Actor and Contribution rules in the grammar. The reference to IntentionalElement in the Contribution rule (indicated with square brackets) is shown as an association in the metamodel. The compositions with LongName in the Actor and Contribution rules as well as the composition with IntentionalElement for the Actor rule (shown without square brackets) are shown as compositions in the metamodel. The unassigned rule call for ElementLink in the grammar corresponds to inheritance in the metamodel. The correlation attribute is mapped to a Boolean attribute (because of the `?=`) and the contribution to the ContributionType enumeration (since ContributionType is defined as an enum in the grammar).

Apart from these features, Xtext also offers below mentioned runtime concepts that allow further customization of the specified textual language.

*Linking:* This concept helps create cross-references related to already existing model elements. It is essentially a link to another grammar rule specifying a non-terminal. When a model element is declared, it can serve as a target of a cross-reference and the usage of the variable is the cross-reference. The TURN grammar uses linking to cross-reference model elements using a `QualifiedName` where the cross-reference is defined using a vertical bar (“|”) between the referenced model element and `QualifiedName` (see reference to `IntentionalElement` from `Dependency` rule in Listing 9), i.e., the model element can be referenced using the rule defined for `QualifiedName` (see Listing 10).

**Listing 9: Linking example**

```
Dependency:  
    (name=QualifiedName longName=LongName)?  
    'dependsOn' dest=[IntentionalElement | QualifiedName];
```

**Listing 10: Rule for defining `QualifiedName` for an `IntentionalElement`**

```
QualifiedName qualifiedName(IntentionalElement intElem) {  
  
    QualifiedName intElemName =  
        QualifiedName.create(EcoreUtil2.getContainerOfType(intElem, Actor.class)  
            .getName(), intElem.getName());  
  
    return intElemName;  
}
```

*Scoping:* This is used to define visibility boundaries of referenced targets for a particular cross-reference. The boundaries can be defined not only inside the file where the cross-reference is present but also over multiple files. One example of scoping is provided in Listing 11 where `ReferencedBoundElement` is referred using its name, if the reference is made in the “elem:

ReferencedBoundElement” attribute of BoundElement. Listing 12 gives another example. In case of map Default, the component *Agent* binds elements such as startPoint and continue. In order to ensure that the component refers to the startPoint of map Default and not the startPoint of map TCS, scoping is defined in a way that the startPoint can only be referred from inside the map Default. Scoping is defined for EndPoints, StartPoints, ReferencedEnds and the three types of ElementLinks.

**Listing 11: TURN scoping mechanism**

```
if (reference == TurnPackage.eINSTANCE.boundElement_Elem) {
    return Scopes.scopeFor(startPoints, [StartPoint e|
    return QualifiedName.create(e.name)
    ], Scopes.scopeFor(endPoints, [EndPoint e|
    return QualifiedName.create(e.name)
    ], Scopes.scopeFor(resps, [RespRef e|
    return QualifiedName.create(e.name)
    ], Scopes.scopeFor(components, [ComponentRef e|
    return QualifiedName.create(e.name)
    ], Scopes.scopeFor(stubs, [Stub e|
    return QualifiedName.create(e.name)
    ], IScope.NULLSCOPE))))))
}
```

**Listing 12: TURN scoping mechanism**

```
map Default {
    start startPoint -> end continue.
    parent Agent: startPoint, continue
}

map TCS #"Terminating Call Screening (TCS)" {
    start startPoint -> X checkTCS -> or {
        [!onTCSlist] -> end success.
        [onTCSlist] -> end failPoint.
    }
    parent Agent: startPoint..success, failPoint, TCSCreeningList
    team TCSCreeningList: checkTCS
}
```

*Validation:* This concept serves to check whether the model fulfills some given features that cannot be defined through grammar rules, e.g., definition of a specific number of elements in

the model, an order of elements in the model, uniqueness of names of model elements etc. For the TURN model, one of the model elements for which we use validation is AndFork where we validate that an AndFork needs to have at least two pathbodies (see Listing 13). Validation rules also exist to ensure unique names of UCMmaps, Concerns, and all GRL elements.

**Listing 13: TURN validation for AndFork**

```
@Check
def checkAndForkHasAtleastTwoPathBodies(AndFork andFork) {
    if (andFork.pathbody.size < 2) {
        error ("And Fork should have at least two paths",
            TurnPackage.Literals.AND_FORK__PATHBODY)
    }
}
```

### 3.4.2 TURN Grammar

The full implementation of the textual syntax of URN in Xtext is available as an appendix to this thesis (see Appendix A). The presented TURN grammar results in the TURN metamodel introduced in Section 3.2 according to the Xtext framework discussed in Section 3.4.1 and supports the concrete syntax for which examples were given in Section 3.3.

One point to mention about the TURN grammar is that some parts of the grammar are left-recursive. Since the Xtext parser reads the grammar top down and left to right, it would endlessly call rules in these situations without consuming any characters. In order to avoid this left recursion, Xtext allows us to left-factor our grammar, i.e., it allows us to define an implicit precedence for a rule over other rules. One example of a left-recursive grammar is given below where “Expression '+' Expression” is left recursive and hence, not allowed in Xtext grammar:

```
Expression: Expression '+' Expression | INT;
```

After left-factoring, we now have two rules instead of one and there is a certain delegation pattern involved. The rule `Expression` does not call itself anymore but `TerminalExpression` instead and the later the rule is called the higher is its precedence.

```
Expression: TerminalExpression ('+' TerminalExpression)*;  
TerminalExpression: INT;
```

In the TURN grammar, left-factoring is used in an `OrFork` body, i.e., `OrBody` where precedence is given to the rule `RegularOrFork` over rule `PathBody` in the delegation chain (see rule `OrBody` in Appendix A). Another part of the TURN grammar where left-factoring is used is the rule `PathBody` (see Appendix A).

### ***3.5 Summary***

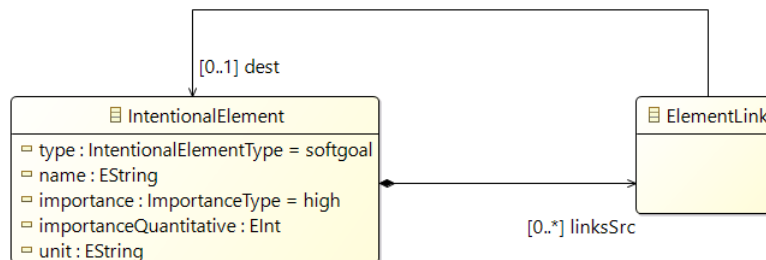
This chapter compares the TURN metamodel with the URN metamodel and shows that there is not much difference in the GRL (Goal-oriented Requirement Language) part of URN and TURN, while there are significant differences in the UCM (Use Case Map) part. It then presents the TURN metamodel and examples of concrete syntax corresponding to the graphical URN examples from Chapter 2. Finally, it describes the Xtext framework and its key features that are used to specify the grammar of the Textual URN presented in Appendix A. The next chapter describes the transformation rules for transforming the Textual URN into the already existing graphical URN.

## CHAPTER 4: TURN to URN Transformation

This chapter describes the transformation process to transform TURN elements into corresponding URN elements. First, required backward links are discussed, followed by the actual transformation rules.

### 4.1 Backward Links

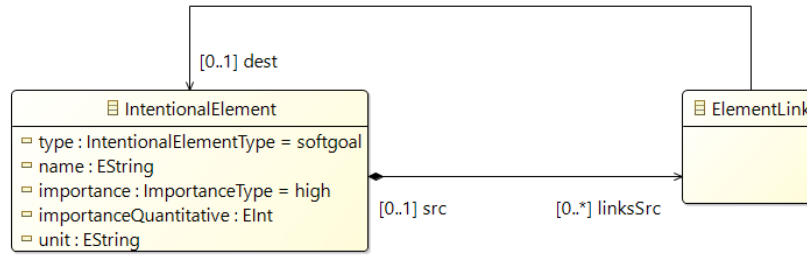
Unlike the URN metamodel, the TURN metamodel automatically created by Xtext based grammar (see Appendix A) does not contain backward links from elements that are contained by other elements inside a model. However, these links are necessary in cases where the parent of an element needs to be known and linked during a transformation. Hence, before proceeding with the transformation of the XMI model specified with Xtext, there is a need to create the required backlinks in the .ecore metamodel generated from the Xtext grammar. These backlinks need to be created only once and can be used in transformation rules to connect the two elements by creating node connections.



**Figure 29: No Backlink from ElementLink to IntentionalElement**

One of the examples where a backlink is required is from an ElementLink to its parent IntentionalElement in order to ensure that the IntentionalElement is not of type Belief. If the IntentionalElement is of type Belief, the transformation must create a BeliefLink instead of an

ElementLink in URN (see Figure 29: No Backlink from ElementLink to IntentionalElement” in Figure 29 and “src” backlink in Figure 30).



**Figure 30: Backlink from ElementLink to IntentionalElement**

## 4.2 Transformation Rules

Below tables show a list of all elements that were transformed from TURN to URN, including the type and the description of transformation rules that are applied to these elements. These rules are implemented using the ATL tool [5] and require an input TURN file saved in XMI form. The rule types are explained in more detail in Section 5.1.1.

**Table 1 Transformation for URNspec and ConcreteURNspec**

TURN	URN	Type of Rule	TURN to URN
URNspec	URNspec, URNdefinition, GRLspec, UCMspec, GRLGraph	Matched Rule	<ol style="list-style-type: none"> <li>1. A new URNdefinition is created along with URNspec, GRLspec, UCMspec and GRLGraph.</li> <li>2. The GRLGraph, UCMmaps and Concerns created in URN are added to URNdefinition.</li> <li>3. Actors, Intentional Elements and Element Links created in URN are added to the GRLspec.</li> <li>4. StrategyGroups, Strategies, Contribution Context Groups, Contribution Contexts and Indicator Conversions created in URN are added to the GRLspec.</li> <li>5. IntentionalElementRefs, Beliefs, ActorRefs, LinkRefs and BeliefLinks created in URN are added to the GRLGraph.</li> </ol>
Concrete URNspec	URNspec	Matched Rule	<ol style="list-style-type: none"> <li>1. Merged into URNspec.</li> </ol>

**Table 2: Transformation for Metadata, Concern and URNLink**

<b>TURN</b>	<b>URN</b>	<b>Type of Rule</b>	<b>TURN to URN</b>
Metadata	Metadata	Matched Rule	1. All the attributes of TURN Metadata are mapped to the attributes of URN Metadata.
Concern	Concern	Matched Rule	1. All the elements, condition and metadata from TURN Concern are mapped to the corresponding attributes in URN Concern.
URNLink	URNLink	Matched Rule	1. Elements of TURN URNLink are transformed directly into elements of URN URNLink (see Listing 14).

The id element of URNspec is initialized to “nextGlobalID” if it exists in metadata, otherwise to zero. All URN elements are assigned the next possible id using an incrementor (see Listing 16) during their transformation. The elements that do not have their counterparts in TURN are assigned a name using their type concatenated with their id, e.g., ActorRef24.

**Table 3: Transformation for Actors, IntentionalElements and ElementLinks**

<b>TURN</b>	<b>URN</b>	<b>Type of Rule</b>	<b>TURN to URN</b>
Actor	Actor, ActorRef	Matched Rule	<ol style="list-style-type: none"> <li>1. A new Actor is created along with an ActorRef.</li> <li>2. The IntentionalElemRefs created in URN are added to the nodes of ActorRef and IntentionalElements are moved to the GRLspec of URN.</li> </ol>
Intentional Element	Intentional Element, Intentional ElementRef, Belief	Matched Rule	<ol style="list-style-type: none"> <li>1. An IntentionalElement is created along with an IntentionalElementRef for each of the IntentionalElements in TURN except of type Belief.</li> <li>2. The IntentionalElement of type Belief is transformed into Belief and a new BeliefLink is created.</li> <li>3. All the ElementLinks in TURN IntentionalElements are moved to the GRLspec created in URN.</li> </ol>
Element Link	ElementLink, LinkRef	Matched Rule	<ol style="list-style-type: none"> <li>1. All the ElementLinks – Contribution, Decomposition and Dependency – are transformed into corresponding ElementLinks in URN.</li> <li>2. Additional LinkRef needs to be created along with each ElementLink.</li> </ol>



For all GRL elements that have a definition as well as a reference (i.e., Actor, IntentionalElement, ElementLink), a new definition and a reference is created during transformation to URN if a definition with the same name does not yet exist in the target model. If such a definition exists, only a new reference is created that points to the existing definition. Also, the *longName* of an element, if present in TURN, is mapped to the name of corresponding element definition in URN. If *longName* is not present, then *name* is used. If both of them exist, the *longName* of an element in TURN is mapped to the name of element in URN and the name of the element in TURN is stored in metadata of the element in URN.

**Table 4: Transformation for Strategies and ContributionContexts**

TURN	URN	Type of Rule	TURN to URN
Strategies Group	Strategies Group	Matched Rule	1. All the evaluation strategies in TURN StrategiesGroup are transformed into strategies in URN StrategiesGroup.
Evaluation Strategy	Evaluation Strategy	Matched Rule	1. All the attributes of EvaluationStrategy in TURN are mapped to the corresponding attributes of EvaluationStrategy in URN.
Indicator Conversion	KPIEval ValueSet, Qualitative Mappings	Matched Rule	1. The subclass LinearConversion of IndicatorConversion in TURN is mapped to KPIEvalValueSet element in URN. 2. The subclass QualToQMappings of IndicatorConversion in TURN is mapped to QualitativeMappings element of URN.
QualToQ Mapping	Qualitative Mapping	Matched Rule	1. All the elements/attributes from TURN QualToQMapping are mapped to URN QualitativeMapping.
Evaluation	Evaluation	Matched Rule	1. TURN Evaluation is transformed into URN Evaluation and the attribute kPIEvalValueSet is mapped to the KPIEvalValueSet and QualitativeMapping created in URN.
Contribution Context Group	Contribution Context Group	Matched Rule	1. All the contribution contexts in TURN ContributionContextGroup are transformed into contribution contexts in URN ContributionContextGroup.

TURN	URN	Type of Rule	TURN to URN
Contribution Context	Contribution Context	Matched Rule	1. All the ContributionChanges and included ContributionContexts in TURN are mapped to the corresponding attributes of ContributionContext in URN.
Contribution Change	Contribution Change	Matched Rule	1. All the attributes of ContributionChange in TURN are mapped to the corresponding attributes of ContributionChange in URN.

**Table 5: Transformation for UCMmap**

TURN	URN	Type of Rule	TURN to URN
UCMmap	UCMmap	Matched Rule	<ol style="list-style-type: none"> <li>1. A new UCMmap is created in URN for every UCMmap in TURN.</li> <li>2. It includes the mapping of all pathNodes from TURN to the pathNodes and node connections created in URN.</li> </ol>

Along with a matched rule described in Table 5, this map-map transformation also includes invoking of called rules (see Table 7; see Section 5.1.1 for a more detailed description of matched rules and called rules). These are *createOrFork*, *createAndFork*, *createTimer*, *createEndPoint*, *createStub*, *connectOrForkNodes*, *connectAndForkNodes*, *connectTimerNodes*, *connectStubNodes*, *createNodeConns* (see Listing 15) and *connectNodes* rules and are written for creating and connecting respective path nodes based on path traversal.

**Table 6: Transformation for StartPoint and RespRef**

TURN	URN	Type of Rule	TURN to URN
StartPoint	StartPoint, Node Connection	Matched Rule	<ol style="list-style-type: none"> <li>1. A new StartPoint and succeeding NodeConnection are created in URN.</li> <li>2. Target StartPoint is bound to a Component using backlinks (see Section 4.1).</li> </ol>
RespRef	Responsibility, RespRef, Node Connection	Matched Rule	<ol style="list-style-type: none"> <li>1. A new Responsibility, RespRef and succeeding NodeConnection for the RespRef are created in URN.</li> <li>2. Target RespRef is bound to a Component using backlinks (see Section 4.1).</li> </ol>

Similar to the transformation of GRL elements, a definition and a reference is created for all UCM elements that need to have a definition and a reference (i.e., Responsibility and Component). If a definition with the same name is already present in URN, only a reference is created pointing to that definition.

**Table 7: Transformation for remaining path nodes**

<b>TURN</b>	<b>URN</b>	<b>Type of Rule</b>	<b>TURN to URN</b>
OrFork	OrFork, Node Connection	Called Rule	<ol style="list-style-type: none"> <li>1. A new OrFork is created along with the required number of succeeding NodeConnection elements.</li> <li>2. All the nodes inside the path bodies of TURN OrFork, until a regular end or a referenced end are collected in a node collector and added to the nodes of target UCMmap.</li> <li>3. URN OrFork is bound to a Component using backlinks (see Section 4.1).</li> </ol>
AndFork	AndFork, Node Connection	Called Rule	<ol style="list-style-type: none"> <li>1. A new AndFork is created along with at least two succeeding NodeConnection elements.</li> <li>2. All the nodes inside the path bodies of TURN AndFork, until a regular end or a referenced end are collected in a node collector and added to the nodes of target UCMmap.</li> <li>3. URN AndFork is bound to a Component using backlinks (see Section 4.1).</li> </ol>
Timer	Timer, Node Connection	Called Rule	<ol style="list-style-type: none"> <li>1. A new Timer is created along with at least one succeeding NodeConnection element plus one succeeding NodeConnection element if the timeout path is also specified.</li> <li>2. All the nodes in timer path bodies, until a regular end or a referenced end are collected in a node collector and added to the nodes of target UCMmap.</li> <li>3. The kind attribute of TURN Timer is mapped to waitType attribute of URN Timer.</li> <li>4. URN Timer is bound to a Component using backlinks (see Section 4.1).</li> </ol>

<b>TURN</b>	<b>URN</b>	<b>Type of Rule</b>	<b>TURN to URN</b>
WaitingPlace	WaitingPlace, Node Connection	Matched Rule	<ol style="list-style-type: none"> <li>1. A new WaitingPlace is created along with the succeeding NodeConnection element.</li> <li>2. The kind attribute of WaitingPlace in TURN is mapped to the waitType attribute of WaitingPlace in URN.</li> <li>3. URN WaitingPlace is bound to a Component using backlinks (see Section 4.1).</li> </ol>
Stub	Stub, Node Connection	Called Rule	<ol style="list-style-type: none"> <li>1. A new Stub is created along with the succeeding NodeConnection elements for each outPath in source Stub.</li> <li>2. Target Stub is dynamic if source Stub has two or more plugins.</li> <li>3. Plugin bindings of TURN Stub are mapped to the bindings of URN Stub.</li> <li>4. All the nodes in the stub outpath, until a regular or a referenced are collected in a node collector and added to the nodes of the target UCMmap.</li> <li>5. Target Stub is bound to a Component using backlinks (see Section 4.1).</li> </ol>
Connect	Connect, Node Connection	Called Rule	<ol style="list-style-type: none"> <li>1. A new Connect is created along with the succeeding NodeConnection element.</li> <li>2. An EmptyPoint is also created along with two succeeding NodeConnections in case of an asynchronous connect.</li> <li>3. Target Connect is bound to the same Component as the connected elements.</li> </ol>
FailurePoint	FailurePoint, Node Connection	Matched Rule	<ol style="list-style-type: none"> <li>1. A new FailurePoint is created along with the succeeding NodeConnection element.</li> <li>2. The condition expression of FailurePoint in TURN is mapped to the expression attribute of FailurePoint in URN.</li> <li>3. Target FailurePoint is bound to a Component using backlinks (see Section 4.1).</li> </ol>
OrJoin	OrJoin	Matched Rule	<ol style="list-style-type: none"> <li>1. A new OrJoin and a succeeding NodeConnection element are created in URN.</li> <li>2. Target OrJoin is bound to a Component using backlinks (see Section 4.1).</li> </ol>

<b>TURN</b>	<b>URN</b>	<b>Type of Rule</b>	<b>TURN to URN</b>
AndJoin	AndJoin	Matched Rule	<ol style="list-style-type: none"> <li>1. A new AndJoin and a succeeding NodeConnection element are created in URN.</li> <li>2. Target OrJoin is bound to a Component using backlinks (see Section 4.1).</li> </ol>
EndPoint	EndPoint	Called Rule	<ol style="list-style-type: none"> <li>1. A new EndPoint is created in URN.</li> <li>2. Target EndPoint is bound to a Component using backlinks (see Section 4.1).</li> </ol>
Plugin Binding	Plugin Binding	Matched Rule	<ol style="list-style-type: none"> <li>1. The InBinding and OutBinding elements created in URN are mapped to the in and out attributes of PluginBinding in URN.</li> </ol>
InBinding	InBinding	Matched Rule	<ol style="list-style-type: none"> <li>1. The source InBinding is transformed directly into target InBinding.</li> </ol>
OutBinding	OutBinding	Matched Rule	<ol style="list-style-type: none"> <li>1. The source OutBinding is transformed directly into target OutBinding.</li> </ol>
Component Binding	Component Binding	Matched Rule	<ol style="list-style-type: none"> <li>1. The source ComponentBinding is transformed directly into target ComponentBinding.</li> </ol>
Component Ref	Component, Component Ref	Matched Rule	<ol style="list-style-type: none"> <li>1. A Component along with a ComponentRef are created in URN.</li> <li>2. The child Components are bound to parent Components using backlinks (see Section 4.1).</li> </ol>

### **4.3 Summary**

This chapter describes the need and types of backward links that are created in the metamodel generated by Xtext. It then describes the rules of transformation from TURN to graphical URN. The different phases of the transformation and the required tools are described in the next chapter.

## CHAPTER 5: Validation

This chapter first gives the implementation details of the transformation from TURN to URN including the tools used for transformation. It then discusses the test cases that were implemented to validate the textual specification of URN models and their transformation.

### 5.1 Implementation

#### 5.1.1 ATL (Atlas Transformation Language) Tool

ATL is a model transformation language used to transform a set of source models into target models. The toolkit is developed and maintained by OBEO [37] and AtlanMod [36]. An ATL program is composed of rules that define the matching of source and target model elements, followed by creation and initialization of target model elements. ATL is a declarative-imperative hybrid, i.e., it has both declarative and imperative parts. There are three types of transformation rules in ATL:

1. *Matched Rules (Declarative Programming)*: These are used when we need to create elements of target model using the information of elements of source model. A matched rule is executed once for each matched source element. Most rules required for the TURN-URN transformation are matched rules. The access to elements is done using the Object Constraint Language (OCL) [32]. A matched rule has an input pattern and an output pattern:
  - a. The input pattern consists of a keyword “from”, declaration of input variable and an OCL expression that returns the input element to be transformed (e.g., see Listing 14; *s* is the input variable of type *Turn!URNlink*). It could also

have a filter or an OCL expression that returns only those elements of the source model that satisfy a set of constraints.

**Listing 14: Matched Rule example**

```
rule URNlink2URNlink{  
  from  
    s: Turn!URNlink  
  to  
    tCon: Urn!URNlink (  
      type <- s.type,  
      fromElem <- s.fromElem,  
      toElem <- s.toElem )  
}
```

- b. The output pattern declares to which output elements of the target model the source elements matching the input pattern will be transformed. It consists of a keyword “to”, variable declaration and initialization of target model element (e.g., see Listing 14; tCon is the output variable of type Urn!RUNLink and the lines with <- are the initializations). It is also possible to have more than one element in the output pattern.

2. *Called Rules (Imperative Programming)*: These are particular types of rules that have to be called explicitly in order to be executed. These can generate target model elements and need to be called from either a matched rule or another called rule and the result of a called rule is the last statement executed in the action block (e.g., see *tNodeConn* in Listing 15). For the TURN-URN transformation, called rules are only used for path nodes with branches (i.e., OrFork, AndFork, Timer (because of the timeout branch), Stub, and EndPoint (because of an optional path following an EndPoint with a Connect)).

**Listing 15: Called Rule example**

```
rule createNodeConns(){  
  to  
    tNodeConn: Urn!NodeConnection()  
  do {  
    tNodeConn;  
  }  
}
```

3. *Lazy Rules (Imperative Programming)*: These are rules that have a source model element as a parameter and require a source pattern to be matched while creating the target model element. The TURN-URN transformation in ATL did not require a lazy rule.

Apart from these rules, ATL also has helpers which are either variables or functions used to implement code that can be reused. For the TURN-URN transformation, helper rules for id increments, maps of source and target nodes, nodes collector and node counters exist. One example is presented in Listing 16.

**Listing 16: Helper Rule example**

```
-- This helper creates an increment method for id  
-- CONTEXT: Integer  
-- RETURN: Integer  
helper context Integer def: inc() : Integer = self + 1;
```

### 5.1.2 jUCMNav Tool

jUCMNav [23] is an open-source Eclipse-based graphical editor and an analysis tool for URN described in Section 2.1. It also supports the creation of Feature Models [31] and the visualization of scenarios with Message Sequence Charts [20]. Since the result of the TURN-URN transformation is a .jucm file, it is loaded using the jUCMNav tool. The jUCMNav tool also has an autolayout feature, which is used to layout the transformed URN model.



### 5.1.3 Implementation Methodology

The implementation of the TURN-URN transformation is done in three phases (see Figure 31). The initial phase requires saving of the TURN model in XMI form. The ATL tool is used for the first phase of transformation and the Java programming language for the second phase of the transformation. This decision is made to address specific differences between TURN and URN metamodels and the intricacies involved in the transformation of UCMs as discussed in the next subsections.

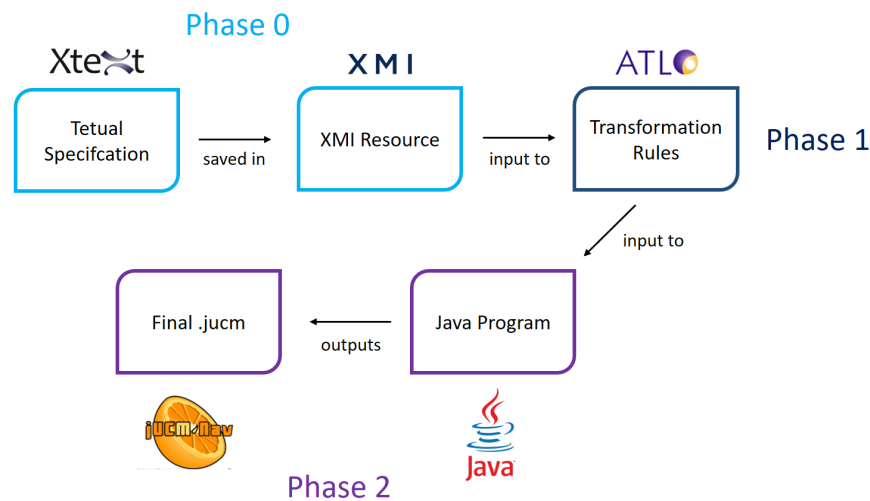


Figure 31: Implementation methodology

#### 5.1.4 Phase 0

The input of this phase is a TURN model written using the Xtext editor. It makes use of the EMF Package Registry, EMF Resource and XMI Resource of the EMF framework to save the TURN model in an XMI file conforming to the .ecore metamodel for TURN automatically generated by Xtext.

#### 5.1.5 Phase 1

The input of this phase is the TURN model in XMI form produced by Phase 0. It involves writing and executing transformation rules (explained in Section 4.2) using the ATL tool and

produces an intermediate .jucm file as the result of the ATL transformation. Note that the ATL tool loads the XMI file with the updated TURN metamodel that includes the backlinks as discussed in Section 4.1, hence making them available to the transformation. This is possible, because backlinks for containment are not explicitly saved in an XMI file, but are rather implied by the nesting of tags in the XMI file (i.e., a tag nested inside a parent tag implies that the class corresponding to the nested tag is contained in the class corresponding to the parent tag).

### 5.1.6 Phase 2

The input of this phase is the URN intermediate file produced by Phase 1. This phase is implemented using the Java programming language and leverages the features provided by EMF. The implementation involves binding of elements to components, creating OrJoins that are implicit in the TURN model, adding Connects to Components, and handling asynchronous Connects with a Timer.

- *Binding elements to components*: This is needed when the TURN model contains ComponentRefs with “from” and “to” attributes, i.e., a ComponentRef is composed of PathNodes that are on the path starting with the PathNode referenced in the “from” attribute (or before the double full stop), up to and including the PathNode referenced in the “to” attribute (or after the double full stop) (e.g., see *Agent* of map TL in Listing 4). The implementation involves traversing through all the PathNodes starting from the first PathNode and consequently binding them to the Component. For traversals that include OrForks or AndForks, the implementation is done in a way that it iterates recursively through all the paths one by one and binds the PathNodes of the path that ends

with the matching “to” element. This needs to be done after all the elements between the “from” element and the “to” element of a map have been created. Since, in ATL, it is difficult to know when the last element of a map gets created as the order in which rules are matched and executed is not deterministic, we decided to do the bindings in the second phase of the transformation after all elements have surely been created (see Figure 32 and Figure 33).

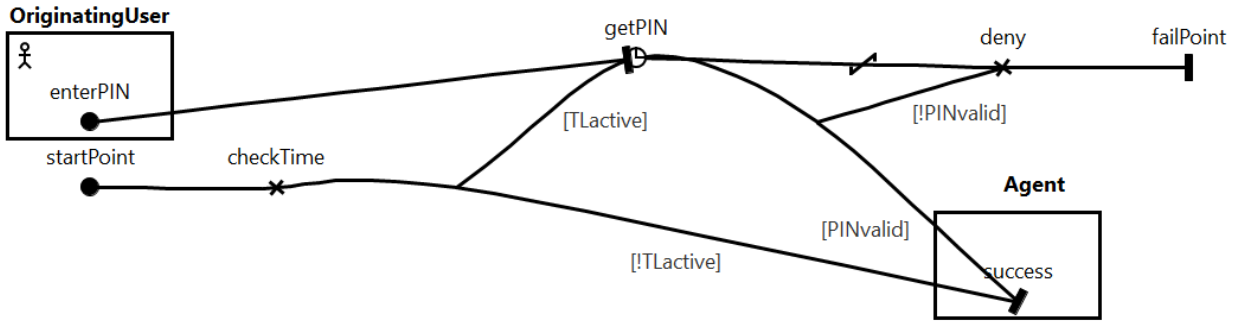


Figure 32: Map TL after Phase 1

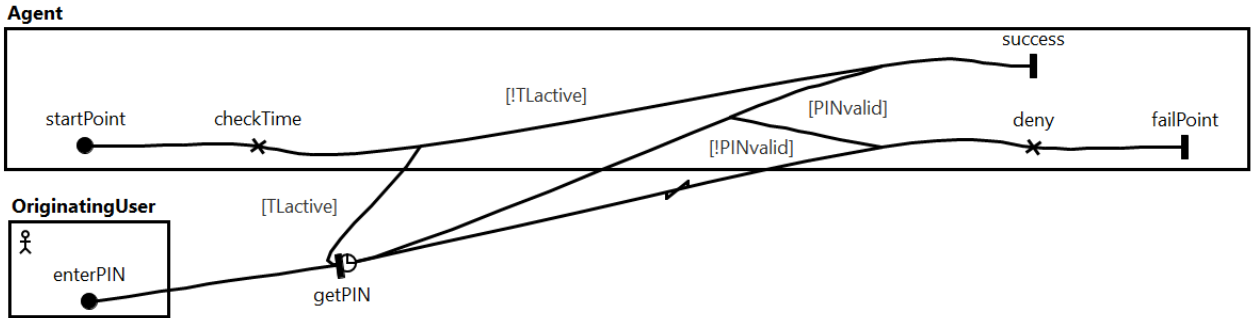


Figure 33: Map TL after binding elements to Components in Phase 2

- *Creating OrJoins*: OrJoins are created when there are implicit OrJoins present in the TURN model, i.e., when EndPoints, FailurePoints or RespRefs are referred by several elements. The implementation validates if an EndPoint, FailurePoint or RespRef has more than two predecessors and hence, creates an OrJoin before the EndPoint/FailurePoint/RespRef in the transformed URN model. The predecessors of the EndPoint/FailurePoint/RespRef are added to the predecessors of the created

OrJoin and the EndPoint/FailurePoint/RespRef itself is added to the successor of the created OrJoin. Lastly, the OrJoin is bound to the Component which contains the EndPoint/FailurePoint/RespRef. This is difficult to achieve using the ATL transformation, because it is not certain whether the definition is reached first during the path traversal or the reference. Since, jUCMNav allows the model to have an EndPoint/FailurePoint/RespRef with more than one predecessor, we decided to take advantage of it and use it to create OrJoins in the next phase of transformation. One example of an implicit OrJoin is presented in map Simple Connection (SC) in Listing 4 in Section 3.3. The ATL transformation results in a URN map with an EndPoint having two predecessors, because *notify* is the EndPoint of two branches (see Figure 34). This map is then fed to the Java program to create an OrJoin before the EndPoint (see Figure 35). Another example is the *success* EndPoint after the ATL transformation shown in Figure 32 and the final result shown in Figure 33.

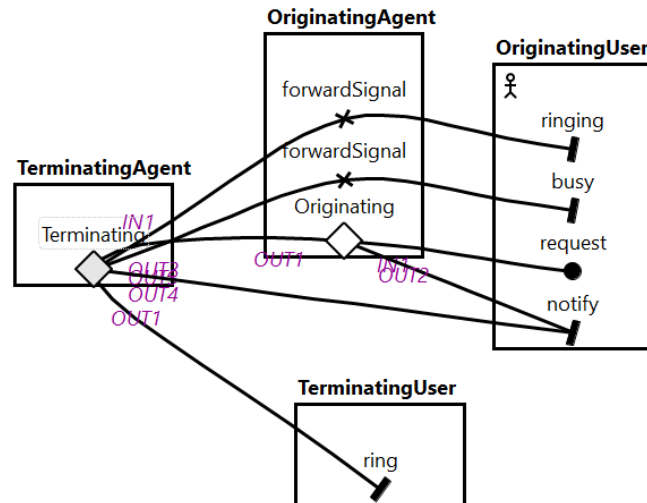


Figure 34: Map SC after Phase 1

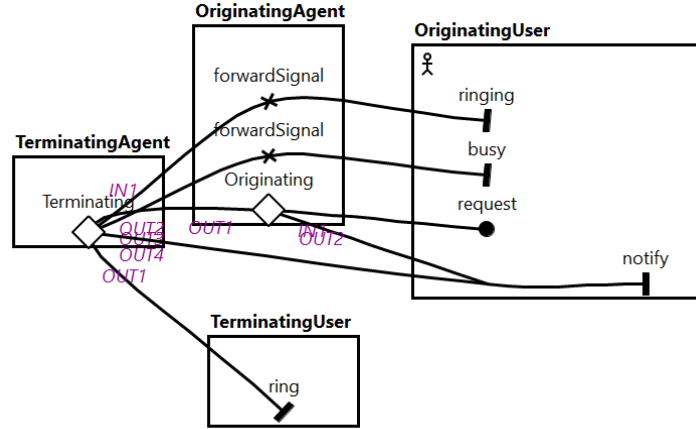


Figure 35: Map SC after Phase 2 (with an OrJoin)

- *Adding Connects to Components:* The implementation involves collecting all PathNodes of type Connect and binding the PathNodes, their predecessors and successors to the matching Component in the transformed URN model. This is again difficult to achieve using ATL transformation as the Connect may occur between “from” and “to” element of a TURN Component and hence, is completed in Phase 2 (see unbound Connect to Timer *getPIN* in Figure 33 and bound Connect in Figure 36).

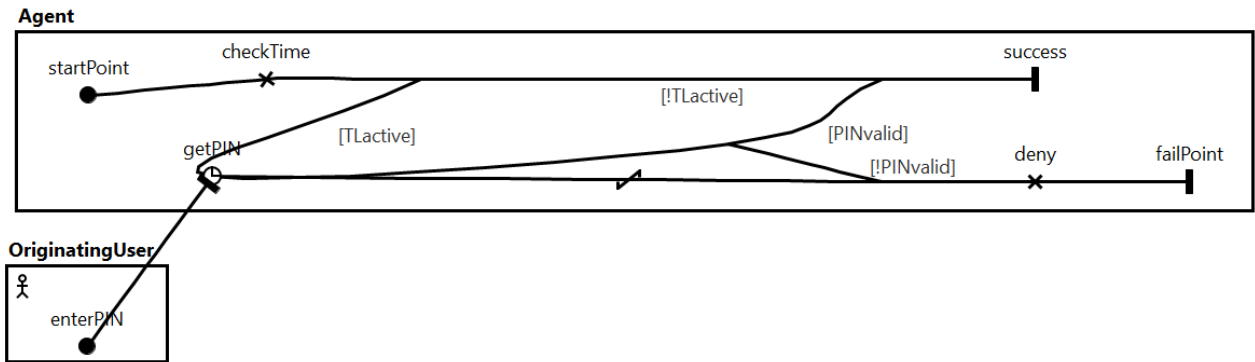
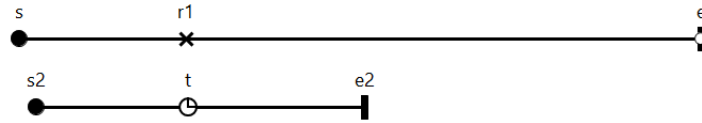


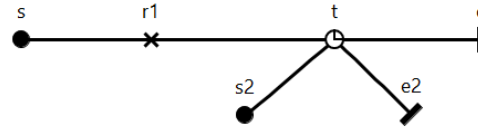
Figure 36: Map TL after Phase 2

- *Handling asynchronous Connects with Timer:* Because a Timer is transformed in Phase 1 with a called rule due to its timeout path, an asynchronous Connect with a Timer needs to be handled in Phase 2, because the Timer may not yet have been

created when the Connect needs to reference it. This is again due to the non-deterministic order in which ATL rules are executed. Consider a UCM model with one path with a StartPoint s2, a Timer t, and an EndPoint e2. A second path has a StartPoint s followed by a RespRef r1, triggers the timer asynchronously, and then ends with an EndPoint e. Therefore, the transformation in Phase 1 adds an EmptyPoint with a name identifying the Timer to the UCM (see Figure 37), which is then connected to the Timer with a proper Connect in Phase 2 (see Figure 38).



**Figure 37: Map with asynchronous Connect after Phase 1**



**Figure 38: Map with asynchronous Connect after Phase 2**

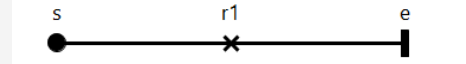
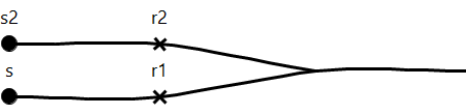

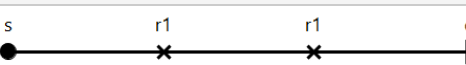

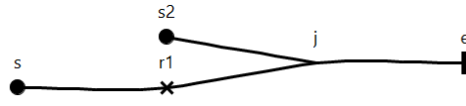
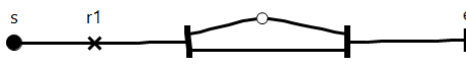
The output of the second phase is a valid .jucm file with a graphical representation of the model which can be loaded using the jUCMNav tool.

## 5.2 Test Cases

In order to validate the implementation of the textual language and its transformation to graphical URN models, test cases are written and executed for each of the (i) 12 path nodes of UCM, (ii) the ComponentRef of UCM, (iii) the combinations of 6 types of GRL nodes and 3 types of GRL links as permitted by the constraints of the URN standard and (iv) the various validation rules. Furthermore, the running example from Figure 1 and Figure 2 (i.e., Listing 1 to Listing 4) is also used as a test and covers additionally Concern, URNlink and Metadata as well

GRL Strategies, Evaluations and ContributionContexts. Representative of all test cases, the Responsibility test cases are listed in this section. All test cases pass. The test cases for Responsibility include the test inputs where a Responsibility is followed by each of the PathBodyNodes, RegularEnds and ReferencedEnds (see Table 8 and Table 9). Another test case exists for the full specification of a Responsibility, i.e., by varying all optional elements a Responsibility can have in TURN. In the case of the Responsibility, there is only one such variation (see Table 10). The test inputs for the remaining UCM path nodes follow the same procedure and are presented in Appendix B. The test cases for ComponentRef, the GRL elements, and the validation rules are also listed in Appendix B. All test cases pass.

**Table 8: Test cases for Responsibility followed by a PathBodyNode or a RegularEnd**

TEST INPUT	TEST OUTPUT
<pre>map ResponsibilityTestEndPoint {   start s -&gt; X r1 -&gt; end e. }</pre>	
<pre>map ResponsibilityTestEndPointAgain {   start s -&gt; X r1 -&gt; end e.   start s2 -&gt; X r2 -&gt; end e. }</pre>	
<pre>map ResponsibilityTestResponsibilityDifferent {   start s -&gt; X r1 -&gt; X r2 -&gt; end e. }</pre>	
<pre>map ResponsibilityTestResponsibilitySame {   start s -&gt; X r1 -&gt; X r1 -&gt; end e. }</pre>	
<pre>map ResponsibilityTestOrFork {   start s -&gt; X r1 -&gt; or {     [condition] -&gt; ;   } -&gt; end e. }</pre>	
<pre>map ResponsibilityTestOrJoin {   start s -&gt; X r1 -&gt; join j -&gt; end e.   start s2 -&gt; j; }</pre>	
<pre>map ResponsibilityTestAndFork {   start s -&gt; X r1 -&gt; and {     * -&gt; ;     * -&gt; ;   } -&gt; end e. }</pre>	

TEST INPUT	TEST OUTPUT
<pre>map ResponsibilityTestAndJoin {   start s -&gt; X r1 -&gt; sync sy -&gt; end e.   start s2 -&gt; sy; }</pre>	
<pre>map ResponsibilityTestWaitingPlace {   start s -&gt; X r1 -&gt; wait -&gt; end e. }</pre>	
<pre>map ResponsibilityTestWaitingPlaceWithWaitKind {   start s -&gt; X r1 -&gt; transient wait -&gt; end e. }</pre>	
<pre>map ResponsibilityTestTimer {   start s -&gt; X r1 -&gt; timer {} -&gt; end e. }</pre>	
<pre>map ResponsibilityTestTimerWithWaitKind {   start s -&gt; X r1 -&gt; persistent timer {}   -&gt; end e. }</pre>	
<pre>map ResponsibilityTestFailurePoint {   start s -&gt; X r1 -&gt; fail -&gt; end e. }</pre>	
<pre>map ResponsibilityTestStub {   start s -&gt; X r1 -&gt; stub () -&gt; end e. }</pre>	
<pre>map ResponsibilityTestStubWithStubType {   start s -&gt; X r1 -&gt; blocking stub ()   -&gt; end e. }</pre>	
<pre>map ResponsibilityTestAsynchronousConnect StartPoint {   start s -&gt; X r1 -&gt; trigger s2; -&gt; end e.   start s2 -&gt; end e2. }</pre>	
<pre>map ResponsibilityTestAsynchronousConnect WaitingPlace {   start s -&gt; X r1 -&gt; trigger wp; -&gt; end e.   start s2 -&gt; wait wp -&gt; end e2. }</pre>	
<pre>map ResponsibilityTestAsynchronousConnectTimer {   start s -&gt; X r1 -&gt; trigger t; -&gt; end e.   start s2 -&gt; timer t -&gt; end e2. }</pre>	

Table 9: Test cases for Responsibility followed by a ReferencedEnd

TEST INPUT	TEST OUTPUT
<pre>map ResponsibilityTestReferencedResponsibility {   start s -&gt; X r1 -&gt; end e.   start s2 -&gt; X r2 -&gt; r1; }</pre>	



TEST INPUT	TEST OUTPUT
<pre>map ResponsibilityTestReferencedOrJoin {   start s -&gt; join j -&gt; end e.   start s2 -&gt; X r1 -&gt; j; }</pre>	
<pre>map ResponsibilityTestReferencedAndJoin {   start s -&gt; sync sy -&gt; end e.   start s2 -&gt; X r1 -&gt; sy; }</pre>	
<pre>map ResponsibilityTestReferencedFailurePoint {   start s -&gt; fail f -&gt; end e.   start s2 -&gt; X r1 -&gt; f; }</pre>	
<pre>map ResponsibilityTestReferencedStub {   start s -&gt; stub st () -&gt; end e.   start s2 -&gt; X r1 -&gt; in st 1; }</pre>	
<pre>map ResponsibilityTestReferencedStubInpath2 {   start s -&gt; stub st () -&gt; end e.   start s2 -&gt; X r1 -&gt; in st 2; }</pre>	

Table 10: Test cases for full specification of Responsibility

TEST INPUT	TEST OUTPUT
<pre>map ResponsibilityTestFull {   start s -&gt; X r#"Responsibility" -&gt; end e. }</pre>	

The test cases for the validation rules cover that an AndFork must have at least 2 pathbodies (i.e., branches; see *checkAndForkHasAtleastTwoPathBodies* in Listing 13) and the validation rules for ensuring the uniqueness of names of UCMmap, Concerns, and all GRL elements.

### 5.3 Summary

This chapter provides the validation for textual specification (TURN) and their transformation from URN. It first describes the tools used for transformation followed by the three phases of the implementation. It then presents the validation rules and test cases including the test inputs and the test outputs. Related work in terms of textual representation and tools used for transformation is discussed in the following chapter.

## CHAPTER 6: Related Work

There exist various solutions to define concrete syntaxes of DSLs. One of the most popular approaches is HUTN [33], specified as a standard by OMG. It defines a generic concrete syntax, which aims to conform to human-usability criteria. It requires a parser generator and the grammar is automatically generated. An obvious advantage of this approach is that any model can be represented in textual notation at a very low cost. However, HUTN imposes very strict constraints on the notation and is a bit verbose. Users cannot provide their own syntax customizations. We used Xtext to define a custom concrete syntax for URN metamodel where concrete syntax for GRL is somewhat similar to HUTN but UCM follows a different approach to define a more compact concrete syntax. Other tools that can be used to define a custom concrete syntax are TCS [22] which uses specifications provided by users to automatically generate editors and tools for model-to-text and text-to-model transformations; and TEF [35] which allows defining multiple syntactic constructs for the same metamodel element but requires the user to specify both the grammar and metamodel, which is redundant.

Several languages support textual and graphical syntaxes including Specification and Description Language (SDL) [18], Message Sequence Charts (MSC) [20], and the Testing and Test Control Notation (TTCN-3) [19], standardized by International Telecommunication Union - Telecommunication Standardization Sector (ITU-T). An effort was taken earlier to create a textual syntax for the already existing Goal-oriented Requirement Language (GRL) [2]. It discusses the challenges faced during the creation of the textual syntax and the conflicts between the reuse of the existing metamodel and the usability of the textual syntax. The experience for Use Case Maps with TURN is similar to the experience with GRL, even more pronounced as the

differences of the TURN metamodel for UCM compared to the URN metamodel for UCM are much greater than the differences of the TURN metamodel for GRL compared to the URN metamodel for GRL.

Another popular example of a textual language is Umple [14][15], that integrates the concepts of UML with programming languages such as Java [14] and PHP [14]. Umple models are written using human-readable text seamlessly integrated with algorithmic code and can also be visualized with the UML notation. This model-is-the-code approach helps developers maintain and evolve the code as the system matures [15]. It uses its own metamodel and grammar.

There are several tools other than ATL that offer model to model transformations. One of the popular implementations is QVT [28], a standard defined by OMG that supports bidirectional transformations. There are two extensions for QVT called QVTd (Declarative) and QVTo (Operational/Procedural). ATL, being a declarative and imperative hybrid, is more expressive with the ability to express any kind of transformations. ATL also executes faster than QVT in most of the cases. Another popular language for M2M transformation is ETL [25] which is built on top of a common expression language (EOL [26]). Epsilon [24] integrates all development tools including editors, code templates and graphical tools for configuring, running and profiling Epsilon programs. ETL can transform many inputs to many output models, and can query/modify both source and target models. Other languages include JTL (Janus Transformation Language [9]), a bidirectional model transformation language specifically designed to support non-bijective transformations and change propagation; Kermeta [13], that borrows concepts from languages such as MOF, OCL, QVT, and BasicMTL [39], and is easier

to learn due to its java-like syntax; and AToMPM [6], a web based modeling environment which provides model transformations based on T-Core [34], a minimal collection of model transformation operators. This has the advantage of executing automatically any custom-built rule-based transformation language.

## CHAPTER 7: Conclusions and Future Work

This thesis proposes a textual notation for an already existing graphical User Requirements Notation (URN). The main objective of the Textual User Requirements Notation (TURN) is to support the modeling of very large URN specifications, where scalability issues have been encountered with the graphical syntax. Thousands of separate graphs and maps become unwieldy to navigate. Hence, TURN allows modelers to create models in a more efficient, faster and platform-independent way. The grammar for Textual URN is written using the Xtext Framework that is a part of the Eclipse Modeling Project. It creates a metamodel, parser and editor from the definition. The editor is used to create models while supporting syntax checking, highlighting and code completion. It also has the ability to provide context-sensitive constraints on the grammar using a Java-based validation language.

Some concepts of the existing URN metamodel do not apply very well to a textual specification, and hence the metamodel for the Textual URN differs for a few significant parts of the URN standard. Most notably, the concept of NodeConnections in a UCM map does not exist in TURN and instead the notion of Path is used. Consequently, to bridge the differences between the URN and TURN metamodels, this thesis also proposes transformation rules for generating a graphical model (URN) from the textual model implemented using the TURN grammar. The input to these rules is a textual model saved in XMI form and the output is an intermediate .jucm file. These rules are implemented using an Eclipse plugin called ATL that is a declarative-imperative hybrid transformation tool. A second phase takes the output of the ATL tool and performs some post-processing (binding elements to their respective Components, creating

implicit OrJoins, and handling asynchronous Connects with Timers) to yield the final URN model that can be visualized using the jUCMNav tool.

Currently, not all concepts from the URN standard are supported by TURN. In future work, we will examine how to best specify the remaining concepts that are not supported yet, e.g., it may be possible to model performance annotations, scenarios and the results of goal model and scenario model analysis, using a textual notation. This will allow modelers to use TURN extensively without depending on the graphical URN for any of the supported concepts. In addition, feature models, a popular extension of URN, could also be integrated into TURN.

Furthermore, the current transformation covers only the direction from TURN to URN but not vice-versa. Also, there is an issue of losing some information when transforming from URN to TURN as URN includes concepts that are required for the graphical syntax but are not needed for the textual syntax. In future, we will extend the transformation so that a URN graphical model can also be transformed into a textual model while saving the information related to graphical syntax in metadata, and hence, improve the consistency of textual and graphical models.

Moreover, the introduced TURN grammar and transformation from TURN to URN can be packaged into one application so that the entire process of creating textual models and the three phases of transformation becomes automated and requires the least manual effort. Currently, the textual model specified using the Xtext editor needs to be saved in XMI using a small helper Java program. This .xmi file is then sent as an input to the transformation program written using the ATL tool (.atl). The output generated by the ATL transformation is then sent as

an input to another Java program for post-processing. This Java program produces a final .jucm file which is a complete graphical URN model.

## REFERENCES

- [1] Abstract Syntax Tree – [http://en.wikipedia.org/wiki/Abstract\\_syntax\\_tree](http://en.wikipedia.org/wiki/Abstract_syntax_tree)
- [2] Abdelzad, V., Amyot, D., and Lethbridge, T.C., “Adding a Textual Syntax to an Existing Graphical Modeling Language: Experience Report with GRL”, *17th Int. Conf. on System Design Languages (SDL 2015)*, Berlin, Germany, October. LNCS 9369, Springer, 159-174, 2015. DOI: 10.1007/978-3-319-24912-4\_12.
- [3] Amyot, D. and Eberlein, A., “An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development”, *Telecommunication Systems Journal*, 24(1):61–94, 2003.
- [4] Amyot, D. and Mussbacher, G., “User Requirements Notation: The First Ten Years, The Next Ten Years”, *Journal of Software (JSW)*, 6(5):747–768, 2011. DOI: 10.4304/jsw.6.5.747-768.
- [5] Atlas Transformation Language tool – [www.eclipse.org/atll/](http://www.eclipse.org/atll/)
- [6] AToMPM tool – <https://acom.cs.mcgill.ca/trac/AToMPM/>
- [7] Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J., “Tropos: An agent-oriented software development methodology”, *Autonomous Agents and Multi-Agent Systems*, 8(3): 203–236, 2004.
- [8] Chung, L., Nixon, B.A., Yu, E., and Mylopoulos, J., “Non-Functional Requirements in Software Engineering”, Kluwer Academic Publishers, 2000.
- [9] Cicchetti, A., Di Ruscio, D., Eramo, R. and Pierantonio, A., “JTL: a bidirectional and change propagating transformation language”, *Software Language Engineering (SLE 2010)*, LNCS 6563, Springer, 183-202, 2011.
- [10] Duran, M.B. and Mussbacher, G., “Investigation of Feature Run-Time Conflicts on Goal Model-Based Reuse”, *Information Systems Frontiers (ISF)*, Springer 18(5):855-875, 2016. DOI: 10.1007/s10796-016-9657-7.
- [11] Duran, M.B., Mussbacher, G., Thimmegowda, N., and Kienzle, J., “On the Reuse of Goal Models”, *17th International System Design Languages Forum (SDL 2015)*, Springer, LNCS 9369:141–158, 2015. DOI: 10.1007/978-3-319-24912-4\_11.
- [12] Eclipse Modeling Framework Project – <http://www.eclipse.org/modeling/emf>
- [13] Fleurey, F., Drey, Z., Vojtisek, D., Faucher, C. and Mahé, V., “Kermeta Language, Reference Manual”, IRISA, 2006 – <http://www.kermeta.org/docs/KerMeta-Manual.pdf>
- [14] Forward, A., et al., “Model-driven rapid prototyping with Umple”, *Softw. Pract. Exper.* 42(7), 781–797, 2012.
- [15] Garzón, M., Aljamaan, H.I., Lethbridge, T.C., “Umple: A Framework for Model Driven Development of Object-Oriented Systems”, *SANER 2015*, pp. 494–498, IEEE CS, 2015.
- [16] Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., and Völkel, S., “Text-based modeling”, *4th International Workshop on Software Language Engineering*, 2007.



- [17] International Telecommunication Union, “Recommendation Z.151 (10/12), User Requirements Notation (URN) – Language definition”, 2012. <http://www.itu.int/rec/T-REC-Z.151/en>
- [18] International Telecommunication Union, “Recommendation Z.100 (12/11), Specification and Description Language – Overview of SDL-2010”, 2011. <http://www.itu.int/rec/T-REC-Z.100-201112-I>
- [19] International Telecommunication Union, “Recommendation Z.161 (11/14), Testing and Test Control Notation Version 3: TTCN-3 Core Language”, 2012. <http://www.itu.int/rec/T-REC-Z.161-201411-I>
- [20] International Telecommunication Union, “Recommendation Z.120 (04/04), Message Sequence Chart (MSC)”, April 2004.
- [21] Jackson, D., “Alloy: a lightweight object modelling notation”, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.
- [22] Jouault, F., Bézivin, J., Kurtev, I., “TCS: a DSL for the specification of textual concrete syntaxes in model engineering”, *GPCE 2006*, pp. 249–254, ACM Press, 2006.
- [23] jUCMNav website, version 7.0, University of Ottawa. <http://jucmnav.softwareengineering.ca/jucmnav>
- [24] Kolovos, D. S., Paige, R. F. and Polack, F. A., “Eclipse development tools for epsilon”, *Eclipse Summit Europe, Eclipse Modeling Symposium*, 2006.
- [25] Kolovos, D. S., Paige, R. F. and Polack, F. A., “The epsilon transformation language”, *Theory and practice of model transformations*, Springer: 46-60, 2008.
- [26] Kolovos, D.S., Paige, R.F., Polack, F.A.C., “The Epsilon Object Language (EOL)”, Rensink, A., Warmer, J. (eds.) *ECMDA-FA 2006*, LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg, 2006.
- [27] Kosslyn, S.M. and Pomerantz, J.R., “Imagery, propositions, and the form of internal representations”, *Cognitive Psychology*, 9(1):52–76, 1977.
- [28] Kurtev, I., “State of the art of QVT: A model transformation language standard”, *Applications of graph transformations with industrial relevance*, Springer: 377-393, 2008.
- [29] Lamsweerde, A.V., “Goal-Oriented Requirements Engineering: A Guided Tour”, *Fifth IEEE International Symposium on Requirements Engineering*, Toronto, Ont., 2001, pp. 249-262. doi: 10.1109/ISRE.2001.948567
- [30] Lamsweerde, A.V., “Requirements Engineering: From System Goals to UML Models to Software Specifications”, John Wiley & Sons Ltd, 2009.
- [31] Liu, Y., Su, Y., Yin, X., and Mussbacher, G., “Combined Propagation-Based Reasoning with Goal and Feature Models”, *4th International Model-Driven Requirements Engineering Workshop (MoDRE 2014)*, Karlskrona, Sweden, IEEE CS, 27-36, 2014. DOI: 10.1109/MoDRE.2014.6890823
- [32] OMG, “UML 2.0 Object Constraint Language specification”, 2006.
- [33] OMG, “UML Human-Usable Textual Notation (HUTN)”, Version 1.0, formal/2004-08-01 (2004), <http://www.omg.org/spec/HUTN/1.0/>
- [34] Syriani, E., Vangheluwe, H., LaShomb, B., “T-Core: A Framework for Custom-built Transformation Engines”, *Software & Systems Modeling*, 14(3):1215–1243, 2015.

- [35] Textual Editing Framework – <http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/index.html>
- [36] The AtlanMod team (Ecole des Mines de Nantes & INRIA) – <http://www.emn.fr/z-info/atlanmod>
- [37] The Obeo company – <http://www.obeo.fr>
- [38] URN Metamodel – <http://jucmnav.softwareengineering.ca/foswiki/ProjetSEG/URNMetaModel>
- [39] Vojtisek, D., “BasicMTL Realization Guide. Inside the Carroll Research Program and part of the MOTOR project”, Technical Report, February 2004.  
[http://modelware.inria.fr/article.php3?id\\_article=45](http://modelware.inria.fr/article.php3?id_article=45)
- [40] Xtext Framework – <https://www.eclipse.org/Xtext/>
- [41] Yu, E., “Modeling Strategic Relationships for Process Reengineering”, Ph.D. thesis, Department of Computer Science, University of Toronto, Canada, 1995.

## Appendix A

### XText Grammar for Textual User Requirements Notation (TURN)

```
grammar org.xtext.project.turn.Turn with org.eclipse.xtext.xbase.Xbase
generate turn "http://www.xtext.org/project/turn/Turn"
import "http://www.eclipse.org/emf/2002/Ecore" as.ecore
```

URNspec:

```
'urnModel' name=QualifiedName
info=(ConcreteURNspec)? &
actors+=Actor* &
(showAsMeansEnd?='showAsMeansEnd')? &
stratGroups+=StrategiesGroup* &
strategies+=EvaluationStrategy* &
indConversions+=IndicatorConversion* &
contribContextGroups+=ContributionContextGroup* &
contribContexts+=ContributionContext* &
ucmMaps+=UCMmap* &
concerns+=Concern* &
urnlinks+=URNlink* &
metadata+=Metadata*;
```

ConcreteURNspec: '{'  
    'description' description=STRING  
    'author' author=STRING  
    'created' created=STRING  
    'modified' modified=STRING  
    'version' specVersion=STRING  
    'urnVersion' urnVersion=STRING  
'}';

Concern: 'concern' name=QualifiedName  
    longName=LongName ':'  
    (condition=Condition)?  
    element=[URNmodelElement | QualifiedName]  
    (',' elements+=[URNmodelElement | QualifiedName])\*;

LongName: {LongName} ('#' longname= (ID | STRING));

Condition: '[' expression=Text ']'  
    ('{' (info=ConcreteCondition)? '}')?;

ConcreteCondition: label=STRING  
    description=STRING;

URNlink: 'link' (name=QualifiedName longName=LongName ':')?  
    ('type' type=Text)?  
    fromElem=[URNmodelElement | QualifiedName] '-->  
    toElem=[URNmodelElement | QualifiedName];

Metadata: 'metadata' (elem = [URNmodelElement | QualifiedName ':')?  
    name=ID '=' value=Text;

```

URNmodelElement: ArtificialRule | URNlink | Concern | Actor | IntentionalElement |
Contribution | Decomposition | Dependency | StrategiesGroup | EvaluationStrategy |
LinearConversion | QualToQMappings | ContributionContextGroup | ContributionContext |
UCMmap | StartPoint | RespRef | WaitingPlace | FailurePoint | Stub | OrFork | OrJoin |
AndFork | AndJoin | Timer | ComponentRef;

```

```

Actor: 'actor' name=QualifiedName
      longName=LongName '{'
      ('importance' (importance=ImportanceType |
importanceQuantitative=QuantitativeValue))?
      elems+=IntentionalElement*
      '}'

```

```

IntentionalElement:
      type=IntentionalElementType name=QualifiedName
      longName=LongName '{'
      ('importance' (importance=ImportanceType |
importanceQuantitative=QuantitativeValue))?
      ('unit' unit=STRING)?
      linksSrc+=ElementLink*
      '}'

```

```

ElementLink: Contribution | Decomposition | Dependency;

```

```

Contribution:
      (name=QualifiedName longName=LongName)?
      'contributesTo' dest=[IntentionalElement | QualifiedName]
      (correlation?='correlated')? 'with' (contribution=ContributionType |
quantitativeContribution=QuantitativeValue);

```

```

Decomposition:
      (name=QualifiedName longName=LongName)?
      decompositionType=DecompositionType 'decomposes' dest=[IntentionalElement |
QualifiedNames];

```

```

Dependency:
      (name=QualifiedName longName=LongName)?
      'dependsOn' dest=[IntentionalElement | QualifiedName];

```

```

ArtificialRule: (longName=STRING)?;

```

```

StrategiesGroup: 'strategiesGroup' name=QualifiedName longName=LongName ':'
      evalStrategy=[EvaluationStrategy]
      ('evalStrategies+=[EvaluationStrategy])*;

```

```

EvaluationStrategy: 'strategy' name=QualifiedName
      longName=LongName '{'
      info=(ConcreteStrategy)?
      evaluations+=Evaluation*
      ('includes' includedStrategy=[EvaluationStrategy]
      ('includedStrategies+=[EvaluationStrategy])*)?
      '}'

```

```

ConcreteStrategy: 'author' author=STRING;

```

```

Evaluation: intElement=[IntentionalElement | QualifiedName]
    (exceeds?='exceeding')? (('evaluation' (qualitativeEvaluation=QualitativeLabel |
    evaluation=QuantitativeValue)) | (indicatorEval=IndicatorEvaluation'convertedWith'
    conversion=[IndicatorConversion]));

IndicatorEvaluation: 'real' (realWorldLabel=STRING | realWorldValue=INT);

IndicatorConversion: LinearConversion | QualToQMappings;

LinearConversion:'linearConversion' name=QualifiedName
    longName=LongName{'
    'unit' unit=STRING
    'target' targetValue=INT
    'threshold' thresholdValue=INT
    'worst' worstValue=INT
    '};

QualToQMappings:'mappingConversion' name=QualifiedName
    longName=LongName{'
    'unit' unit=STRING
    mappings+=QualToQMMapping+
    '};

QualToQMMapping: (exceeds?='exceeding')? 'real' realWorldLabel=STRING '-->'
    (qualitativeEvaluation=QualitativeLabel | evaluation= QuantitativeValue);

ContributionContextGroup: 'contributionContextGroup' name=QualifiedName
    longName=LongName ':'
    contrib=[ContributionContext]
    (',' contribs+=[ContributionContext])*;

ContributionContext:'contributionContext' name=QualifiedName
    longName=LongName '{'
    changes+=ContributionChange*
    ('includes' includedContext=[ContributionContext]
    (',' includedContexts+=[ContributionContext])*)?
    '};

ContributionChange: contribution=[Contribution | QualifiedName]
    'with' (newContribution=ContributionType |
    newQuantitativeContribution=QuantitativeValue);

enum QualitativeLabel: denied | weaklyDenied | weaklySatisfied | satisfied | conflict |
unknown | none;

enum IntentionalElementType: softgoal | goal | task | resource | belief | indicator;

enum ContributionType: make | help | somePositive | unknown | someNegative | hurt | break;

enum ImportanceType: high | medium | low | none;

enum DecompositionType: and | or | xor;

QuantitativeValue returns ecore::EInt: ('-'|'+')? INT;

```

```

UCMmap: (singleton?='singleton')? 'map' name=QualifiedName
    longName=LongName '{'
    stubs+=StubDeclaration*
    paths+=Path*
    components+=ComponentRef*
    '}'

Path: startPoint=StartPoint
    pathBody = PathBody;

StartPoint: (('start' (name=QualifiedName longName=LongName)? (precondition=Condition)?) |
    (failKind = FailureKind name=QualifiedName longName=LongName
    ('catches' '[' catches=STRING ']'?)))

FailureKind: 'abort' | 'failure';

PathBody: PathWithRegularEnd;

PathWithRegularEnd returns PathBody: PathWithReferencedEnd (pathEnd=RegularEnd)?;

PathWithReferencedEnd returns PathBody: PathWithReferencedStub (
    (referencedEnd=[ReferencedEnd])? ';' );

PathWithReferencedStub returns PathBody: PathBodyNodes ('in'
    referencedStub=[Stub | QualifiedName]
    index=PositiveInteger ';' );

PathBodyNodes returns PathBody:{PathBodyNodes} Arrow (pathNodes+=PathBodyNode '->');

Arrow: '->';

PathBodyNode: AndJoin | RespRef | OrJoin | WaitingPlace | FailurePoint | Timer | Connect;

RegularEnd: EndpointWithConnect | AndFork | OrFork | Stub;

EndpointWithConnect: regularEnd=EndPoint
    ('->' connect=Connect)?;

ReferencedEnd: RespRef | OrJoin | Stub | AndJoin | FailurePoint;

RespRef: 'X' name = QualifiedName
    longName=LongName;

OrJoin: {OrJoin} 'join' name = QualifiedName
    longName=LongName;

AndJoin: ('synch' | 'sync' | '|') name = QualifiedName
    longName=LongName ;

WaitingPlace: kind=(WAITKIND)? 'wait' (name=QualifiedName longName=LongName)?
    (condition=Condition)? ;

Timer: {Timer} kind=(WAITKIND)? 'timer' (name=QualifiedName longName=LongName)? ('{'
    'timeout'? timeoutCondition=(Condition)? timeoutPath = (PathBody)?
    '}' )? (condition=Condition)?;

```

```

Connect: 'trigger'
    connectsTo= [ConnectElement | QualifiedName] ';;

ConnectElement: WaitingPlace | Timer | StartPoint;

FailurePoint: 'fail' (name = QualifiedName longName=LongName)?
    ((condition=Condition failLabel=FailureLabel) | (failLabel=FailureLabel
    condition=Condition));

EndPoint: 'end' (name = QualifiedName longName=LongName)? (condition=Condition)? '.';

OrFork: 'or' (((name=QualifiedName longName=LongName)? '{'
    body=OrBody
    '})' | condition=Condition) (connectingOrBody=PathBody));

OrBody: ElseOrForkBody;

ElseOrForkBody returns OrBody: OrForkBody ('[else]' elseBody=PathBody)?;

OrForkBody returns OrBody: (regularBody+=RegularOrFork)+;

RegularOrFork: condition=Condition
    pathBody=PathBody;

AndFork: 'and' (name=QualifiedName longName=LongName)? '{'
    ('*'pathbody+=PathBody)*
    '}'(connectingAndBody=PathBody)?;

Stub: (stubType=StubType)? 'stub' ('in' index=PositiveInteger)?
    (((name = QualifiedName longName=LongName)? params=StubParameters) |
    (stubRef=[StubDeclaration | QualifiedName])) ('{'
    outPaths+=StubOutPath*
    '})'? (connectingStubBody=PathBody)?;

StubType: 'synchronizing' | 'blocking';

StubParameters: {StubParameters}
    '(' plugin+=PluginBinding* ')';

StubDeclaration: name=QualifiedName longName=LongName '='
    params=StubParameters;

StubOutPath: ('out' index=PositiveInteger)
    ('threshold' '[' threshold=PositiveInteger ']')?
    path=PathBody;

PluginBinding: (condition=Condition)?
    ('replication' replication=PositiveInteger)?
    map=[UCMmap]
    (':'binding=Binding
    (',' bindings+=Binding)*);

ComponentRef: ('protected')? kind=(COMPONENTKIND)?
    name=QualifiedName longName=LongName
    (':' element= BoundOption (',' elements+= BoundOption)*);

BoundOption: BoundElement | LongBoundElement;

```

```

BoundElement: elem=[ReferencedBoundElement | QualifiedName] ;

ReferencedBoundElement: (ComponentRef | StartPoint | PathBodyNode | EndPoint | RegularEnd);

LongBoundElement: from=(QualifiedName) '..' to=(QualifiedName);

Binding: InBinding | OutBinding | ComponentBinding;

InBinding: mapStart = [StartPoint | QualifiedName] '=' ('in' index=PositiveInteger);

OutBinding: mapEnd = [EndPoint | QualifiedName] '=' ('out' index=PositiveInteger);

ComponentBinding: componentIn=[ComponentRef | QualifiedName] '='
    componentOut=[ComponentRef | QualifiedName];

FailureLabel: '[[ ' failure=STRING ']]';

enum WAITKIND: persistent | transient;

enum COMPONENTKIND: team | object | process | agent | actor | parent;

PositiveInteger returns ecore::EInt: INT;

Text: content+=TextContent+;

TextContent: ID | ('@') | ('#') | '!' | '%' | '&' | '*' | INT;

```



## Appendix B

### Test cases for StartPoint

```
urnModel StartPointTest

// StartPoint followed by a PathBodyNode (Responsibility, OrJoin, AndJoin, WaitingPlace,
// Timer, FailurePoint, Connect)
// or RegularEnd (EndpointWithConnect, OrFork, AndFork, Stub)

map StartPointTestEndPoint {
  start s -> end e.
}

map StartPointTestEndPointAgain {
  start s -> end e.
  start s2 -> end e.
}

map StartPointTestResponsibility {
  start s -> X r1 -> end e.
}

map StartPointTestResponsibilityAgain {
  start s -> X r1 -> end e.
  start s2 -> X r1 -> end e2.
}

map StartPointTestOrFork {
  start s -> or {
    [condition] -> ;
  } -> end e.
}

map StartPointTestOrJoin {
  start s -> join j -> end e.
  start s2 -> j;
}

map StartPointTestAndFork {
  start s -> and {
    * -> ;
    * -> ;
  } -> end e.
}

map StartPointTestAndJoin {
  start s -> sync sy -> end e.
  start s2 -> sy;
}

map StartPointTestWaitingPlace {
  start s -> wait -> end e.
}
```

```

map StartPointTestWaitingPlaceWithWaitKind {
    start s -> transient wait -> end e.
}

map StartPointTestTimer {
    start s -> timer {} -> end e.
}

map StartPointTestTimerWithWaitKind {
    start s -> persistent timer {} -> end e.
}

map StartPointTestFailurePoint {
    start s -> fail -> end e.
}

map StartPointTestStub {
    start s -> stub () -> end e.
}

map StartPointTestStubWithStubType {
    start s -> blocking stub () -> end e.
}

map StartPointTestAsynchronousConnectStartPoint {
    start s -> trigger s2; -> end e.
    start s2 -> end e2.
}

map StartPointTestAsynchronousConnectWaitingPlace {
    start s -> trigger wp; -> end e.
    start s2 -> wait wp -> end e2.
}

map StartPointTestAsynchronousConnectTimer {
    start s -> trigger t; -> end e.
    start s2 -> timer t -> end e2.
}

// StartPoint followed by a ReferencedEnd
// (Responsibility, OrJoin, AndJoin, FailurePoint, Stub)

map StartPointTestReferencedResponsibility {
    start s -> X r1 -> end e.
    start s2 -> r1;
}

map StartPointTestReferencedOrJoin {
    start s -> join j -> end e.
    start s2 -> j;
}

map StartPointTestReferencedAndJoin {
    start s -> sync sy -> end e.
    start s2 -> sy;
}

```

```

map StartPointTestReferencedFailurePoint {
  start s -> fail f -> end e.
  start s2 -> f;
}

map StartPointTestReferencedStub {
  start s -> stub st () -> end e.
  start s2 -> in st 1;
}

map StartPointTestReferencedStubInpath2 {
  start s -> stub st () -> end e.
  start s2 -> in st 2;
}

// full specification of StartPoint

map StartPointTestFull {
  start s#"StartPoint One" -> end e.
}

map StartPointTestNoName {
  start -> end e.
}

map StartPointTestAbort {
  abort s#"StartPoint One" catches ["ExceptionOne"] -> end e.
}

map StartPointTestFailure {
  failure s#"StartPoint One" catches ["ExceptionOne"] -> end e.
}

map StartPointTestCondition {
  start s#"StartPoint One" [condition] -> end e.
}

map StartPointTestOnlyCondition {
  start s [condition] -> end e.
}

```

## Test cases for EndPoint

```
urnModel EndPointTest

// EndPoint followed by a PathBodyNode (not possible: Responsibility, OrJoin, AndJoin,
// WaitingPlace, Timer, FailurePoint, Connect)
// or RegularEnd (not possible: EndpointWithConnect, OrFork, AndFork, Stub)

// EndPoint followed by a ReferencedEnd
// (not possible: Responsibility, OrJoin, AndJoin, FailurePoint, Stub)

// full specification of EndPoint

map EndPointTestFull {
  start s -> end e#"EndPoint One".
}

map EndPointTestNoName {
  start s -> end.
}

map EndPointTestCondition {
  start s -> end e#"EndPoint One" [condition].
}

map EndPointTestOnlyCondition {
  start s -> end e [condition].
}

map EndPointTestSynchronousConnectStartPoint {
  start s -> end e. -> trigger s2;
  start s2 -> end e2.
}

map EndPointTestSynchronousConnectWaitingPlace {
  start s -> end e. -> trigger wp;
  start s2 -> wait wp -> end e2.
}

map EndPointTestSynchronousConnectTimer {
  start s -> end e. -> trigger t;
  start s2 -> timer t -> end e2.
}
```

## Test cases for Responsibility

```
urnModel ResponsibilityTest

// Responsibility followed by a PathBodyNode (Responsibility, OrJoin, AndJoin,
// WaitingPlace, Timer, FailurePoint, Connect)
// or RegularEnd (EndpointWithConnect, OrFork, AndFork, Stub)

map ResponsibilityTestEndPoint {
  start s -> X r1 -> end e.
}

map ResponsibilityTestEndPointAgain {
  start s -> X r1 -> end e.
  start s2 -> X r2 -> end e.
}

map ResponsibilityTestResponsibilityDifferent {
  start s -> X r1 -> X r2 -> end e.
}

map ResponsibilityTestResponsibilitySame {
  start s -> X r1 -> X r1 -> end e.
}

map ResponsibilityTestOrFork {
  start s -> X r1 -> or {
    [condition] -> ;
  } -> end e.
}

map ResponsibilityTestOrJoin {
  start s -> X r1 -> join j -> end e.
  start s2 -> j;
}

map ResponsibilityTestAndFork {
  start s -> X r1 -> and {
    * -> ;
    * -> ;
  } -> end e.
}

map ResponsibilityTestAndJoin {
  start s -> X r1 -> sync sy -> end e.
  start s2 -> sy;
}

map ResponsibilityTestWaitingPlace {
  start s -> X r1 -> wait -> end e.
}

map ResponsibilityTestWaitingPlaceWithWaitKind {
  start s -> X r1 -> transient wait -> end e.
}
```

```

map ResponsibilityTestTimer {
    start s -> X r1 -> timer {} -> end e.
}

map ResponsibilityTestTimerWithWaitKind {
    start s -> X r1 -> persistent timer {} -> end e.
}

map ResponsibilityTestFailurePoint {
    start s -> X r1 -> fail -> end e.
}

map ResponsibilityTestStub {
    start s -> X r1 -> stub () -> end e.
}

map ResponsibilityTestStubWithStubType {
    start s -> X r1 -> blocking stub () -> end e.
}

map ResponsibilityTestAsynchronousConnectStartPoint {
    start s -> X r1 -> trigger s2; -> end e.
    start s2 -> end e2.
}

map ResponsibilityTestAsynchronousConnectWaitingPlace {
    start s -> X r1 -> trigger wp; -> end e.
    start s2 -> wait wp -> end e2.
}

map ResponsibilityTestAsynchronousConnectTimer {
    start s -> X r1 -> trigger t; -> end e.
    start s2 -> timer t -> end e2.
}

// Responsibility followed by a ReferencedEnd
// (Responsibility, OrJoin, AndJoin, FailurePoint, Stub)

map ResponsibilityTestReferencedResponsibility {
    start s -> X r1 -> end e.
    start s2 -> X r2 -> r1;
}

map ResponsibilityTestReferencedOrJoin {
    start s -> join j -> end e.
    start s2 -> X r1 -> j;
}

map ResponsibilityTestReferencedAndJoin {
    start s -> sync sy -> end e.
    start s2 -> X r1 -> sy;
}

```

```

map ResponsibilityTestReferencedFailurePoint {
  start s -> fail f -> end e.
  start s2 -> X r1 -> f;
}

map ResponsibilityTestReferencedStub {
  start s -> stub st () -> end e.
  start s2 -> X r1 -> in st 1;
}

map ResponsibilityTestReferencedStubInpath2 {
  start s -> stub st () -> end e.
  start s2 -> X r1 -> in st 2;
}

// full specification of Responsibility

map ResponsibilityTestFull {
  start s -> X r#"Responsibility" -> end e.
}

```

## Test cases for OrFork

```
urnModel OrForkTest

// OrFork followed by a PathBodyNode (Responsibility, OrJoin, AndJoin, WaitingPlace, Timer,
// FailurePoint, Connect)
// or RegularEnd (EndpointWithConnect, OrFork, AndFork, Stub)

map OrForkTestEndPoint {
  start s -> or {
    [condition] -> ;
  } -> end e.
}

map OrForkTestEndPointAgain {
  start s -> or {
    [condition] -> ;
  } -> end e.
  start s2 -> or {
    [condition] -> ;
  } -> end e.
}

map OrForkTestResponsibility {
  start s -> or {
    [condition] -> ;
  } -> X r1 -> end e.
}

map OrForkTestResponsibilityAgain {
  start s -> or {
    [condition] -> ;
  } -> X r1 -> end e.
  start s2 -> or {
    [condition] -> ;
  } -> X r1 -> end e2.
}

map OrForkTestOrFork {
  start s -> or {
    [condition] -> ;
  } -> or {
    [condition] -> ;
  } -> end e.
}

map OrForkTestOrJoin {
  start s -> or {
    [condition] -> ;
  } -> join j -> end e.
  start s2 -> j;
}
```



```

map OrForkTestAndFork {
  start s -> or {
    [condition] -> ;
  } -> and {
    * -> ;
    * -> ;
  } -> end e.
}

map OrForkTestAndJoin {
  start s -> or {
    [condition] -> ;
  } -> sync sy -> end e.
  start s2 -> sy;
}

map OrForkTestWaitingPlace {
  start s -> or {
    [condition] -> ;
  } -> wait -> end e.
}

map OrForkTestWaitingPlaceWithWaitKind {
  start s -> or {
    [condition] -> ;
  } -> transient wait -> end e.
}

map OrForkTestTimer {
  start s -> or {
    [condition] -> ;
  } -> timer {} -> end e.
}

map OrForkTestTimerWithWaitKind {
  start s -> or {
    [condition] -> ;
  } -> persistent timer {} -> end e.
}

map OrForkTestFailurePoint {
  start s -> or {
    [condition] -> ;
  } -> fail -> end e.
}

map OrForkTestStub {
  start s -> or {
    [condition] -> ;
  } -> stub () -> end e.
}

```

```

map OrForkTestStubWithStubType {
  start s -> or {
    [condition] -> ;
  } -> blocking stub () -> end e.
}

map OrForkTestAsynchronousConnectStartPoint {
  start s -> or {
    [condition] -> ;
  } -> trigger s2; -> end e.
  start s2 -> end e2.
}

map OrForkTestAsynchronousConnectWaitingPlace {
  start s -> or {
    [condition] -> ;
  } -> trigger wp; -> end e.
  start s2 -> wait wp -> end e2.
}

map OrForkTestAsynchronousConnectTimer {
  start s -> or {
    [condition] -> ;
  } -> trigger t; -> end e.
  start s2 -> timer t -> end e2.
}

// OrFork followed by a ReferencedEnd
// (Responsibility, OrJoin, AndJoin, FailurePoint, Stub)

map OrForkTestReferencedResponsibility {
  start s -> X r1 -> end e.
  start s2 -> or {
    [condition] -> ;
  } -> r1;
}

map OrForkTestReferencedOrJoin {
  start s -> join j -> end e.
  start s2 -> or {
    [condition] -> ;
  } -> j;
}

map OrForkTestReferencedAndJoin {
  start s -> sync sy -> end e.
  start s2 -> or {
    [condition] -> ;
  } -> sy;
}

map OrForkTestReferencedFailurePoint {
  start s -> fail f -> end e.
  start s2 -> or {
    [condition] -> ;
  } -> f;
}

```

```

map OrForkTestReferencedStub {
  start s -> stub st () -> end e.
  start s2 -> or {
    [condition] -> ;
  } -> in st 1;
}

map OrForkTestReferencedStubInpath2 {
  start s -> stub st () -> end e.
  start s2 -> or {
    [condition] -> ;
  } -> in st 2;
}

// branches of OrFork with all possible path nodes as first element on branch

map OrForkTestBranches {
  start s -> or {
    [c] -> end e.
    [c1] -> end e1.
    [c2] -> end e1.
    [c3] -> X r1 -> end e2.
    [c4] -> X r1 -> end e3.
    [c5] -> or {
      [condition] -> end e4.
      [!condition] -> end e5.
    }
    [c6] -> join j -> end e6.
    [c7] -> and {
      * -> end e7.
      * -> end e8.
    }
    [c8] -> sync sy -> end e9.
    [c9] -> wait -> end e10.
    [c10] -> transient wait -> end e11.
    [c11] -> timer {} -> end e12.
    [c12] -> persistent timer {} -> end e13.
    [c13] -> fail -> end e14.
    [c14] -> stub () ; -> end e15.
    [c15] -> blocking stub () ; -> end e16.
    [c16] -> trigger s4; -> end e17.
    [c17] -> trigger wp; -> end e18.
    [c18] -> trigger t; -> end e19.
    [c19] -> r2;
    [c20] -> j2;
    [c21] -> sy2;
    [c22] -> f;
    [c23] -> in st2 1;
    [c24] -> in st2 2;
  }
  start s2 -> j;
  start s3 -> sy;
  start s4 -> end e44.
  start s5 -> wait wp -> end e55.
  start s6 -> timer t -> end e66.
  start s7 -> X r2 -> end e77.
  start s8 -> join j2 -> end e88.
}

```

```

    start s9 -> sync sy2 -> end e99.
    start s10 -> fail f -> end e1010.
    start s11 -> stub st2 () ; -> end e1111.
}

map OrForkTestMergedBranches {
    start s -> or {
        [c] -> ;
        [c1] -> X r1 -> ;
        [c2] -> X r1 -> ;
        [c3] -> or {
            [condition] -> ;
            [!condition] -> ;
        } -> ;
        [c4] -> join j -> ;
        [c5] -> and {
            * -> ;
            * -> ;
        } -> ;
        [c6] -> sync sy -> ;
        [c7] -> wait -> ;
        [c8] -> transient wait -> ;
        [c9] -> timer {} -> ;
        [c10] -> persistent timer {} -> ;
        [c11] -> fail -> ;
        [c12] -> stub () ; -> ;
        [c13] -> blocking stub () ; -> ;
        [c14] -> trigger s4; -> ;
        [c15] -> trigger wp; -> ;
        [c16] -> trigger t; -> ;
        [c17] -> r2;
        [c18] -> j2;
        [c19] -> sy2;
        [c20] -> f;
        [c21] -> in st2 1;
        [c22] -> in st2 2;
    } -> end e.
    start s2 -> j;
    start s3 -> sy;
    start s4 -> end e44.
    start s5 -> wait wp -> end e55.
    start s6 -> timer t -> end e66.
    start s7 -> X r2 -> end e77.
    start s8 -> join j2 -> end e88.
    start s9 -> sync sy2 -> end e99.
    start s10 -> fail f -> end e1010.
    start s11 -> stub st2 () ; -> end e1111.
}

// full specification of OrFork

map OrForkTestFull {
    start s -> or o#"OrFork One" {
        [condition] -> ;
    } -> end e.
}

```

```

map OrForkTestTwoBranches {
  start s -> or o#"OrFork One" {
    [condition1] -> ;
    [condition2] -> ;
  } -> end e.
}

map OrForkTestElseBranch {
  start s -> or o#"OrFork One" {
    [condition] -> ;
    [else] -> ;
  } -> end e.
}

map OrForkTestNoConnectingBody {
  start s -> or o#"OrFork One" {
    [condition1] -> end e1.
    [condition2] -> end e2.
  }
}

map OrForkTestRegularBranchAndConnectingBranch {
  start s -> or o#"OrFork One" {
    [condition1] -> end e1.
    [condition2] -> ;
  } -> end e2.
}

map OrForkTestRegularBranchAndTwoConnectingBranches {
  start s -> or o#"OrFork One" {
    [condition1] -> end e1.
    [condition2] -> ;
    [condition3] -> ;
  } -> end e2.
}

map OrForkTestJustCondition {
  start s -> or [condition] -> end e.
}

```

## Test cases for OrJoin

```
urnModel OrJoinTest

// OrJoin followed by a PathBodyNode (Responsibility, OrJoin, AndJoin, WaitingPlace, Timer,
// FailurePoint, Connect)
// or RegularEnd (EndpointWithConnect, OrFork, AndFork, Stub)

map OrJoinTestEndPoint {
  start s -> join j -> end e.
  start s2 -> j;
}

map OrJoinTestEndPointAgain {
  start s -> join j -> end e.
  start s2 -> j;
  start s3 -> join j2 -> end e.
  start s4 -> j2;
}

map OrJoinTestResponsibility {
  start s -> join j -> X r1 -> end e.
  start s2 -> j;
}

map OrJoinTestResponsibilityAgain {
  start s -> join j -> X r1 -> end e.
  start s2 -> j;
  start s3 -> join j2 -> X r1 -> end e2.
  start s4 -> j2;
}

map OrJoinTestOrFork {
  start s -> join j -> or {
    [condition] -> ;
  } -> end e.
  start s2 -> j;
}

map OrJoinTestOrJoin {
  start s -> join j -> join j2 -> end e.
  start s2 -> j;
  start s3 -> j2;
}

map OrJoinTestAndFork {
  start s -> join j -> and {
    * -> ;
    * -> ;
  } -> end e.
  start s2 -> j;
}

map OrJoinTestAndJoin {
  start s -> join j -> sync sy -> end e.
  start s2 -> j;
  start s3 -> sy;
}
```

```

map OrJoinTestWaitingPlace {
  start s -> join j -> wait -> end e.
  start s2 -> j;
}

map OrJoinTestWaitingPlaceWithWaitKind {
  start s -> join j -> transient wait -> end e.
  start s2 -> j;
}

map OrJoinTestTimer {
  start s -> join j -> timer {} -> end e.
  start s2 -> j;
}

map OrJoinTestTimerWithWaitKind {
  start s -> join j -> persistent timer {} -> end e.
  start s2 -> j;
}

map OrJoinTestFailurePoint {
  start s -> join j -> fail -> end e.
  start s2 -> j;
}

map OrJoinTestStub {
  start s -> join j -> stub () -> end e.
  start s2 -> j;
}

map OrJoinTestStubWithStubType {
  start s -> join j -> blocking stub () -> end e.
  start s2 -> j;
}

map OrJoinTestAsynchronousConnectStartPoint {
  start s -> join j -> trigger s3; -> end e.
  start s2 -> j;
  start s3 -> end e2.
}

map OrJoinTestAsynchronousConnectWaitingPlace {
  start s -> join j -> trigger wp; -> end e.
  start s2 -> j;
  start s3-> wait wp -> end e2.
}

map OrJoinTestAsynchronousConnectTimer {
  start s -> join j -> trigger t; -> end e.
  start s2 -> j;
  start s3 -> timer t -> end e2.
}

```

```

// OrJoin followed by a ReferencedEnd
// (Responsibility, OrJoin, AndJoin, FailurePoint, Stub)

map OrJoinTestReferencedResponsibility {
  start s -> X r1 -> end e.
  start s2 -> join j -> r1;
  start s3 -> j;
}

map OrJoinTestReferencedOrJoin {
  start s -> join j2 -> end e.
  start s2 -> join j -> j2;
  start s3 -> j;
}

map OrJoinTestReferencedAndJoin {
  start s -> sync sy -> end e.
  start s2 -> join j -> sy;
  start s3 -> j;
}

map OrJoinTestReferencedFailurePoint {
  start s -> fail f -> end e.
  start s2 -> join j -> f;
  start s3 -> j;
}

map OrJoinTestReferencedStub {
  start s -> stub st () -> end e.
  start s2 -> join j -> in st 1;
  start s3 -> j;
}

map OrJoinTestReferencedStubInpath2 {
  start s -> stub st () -> end e.
  start s2 -> join j -> in st 2;
  start s3 -> j;
}

// full specification of OrJoin

map OrJoinTestFull {
  start s -> join j#"OrJoin One" -> end e.
  start s2 -> j;
}

```



## Test cases for AndFork

```
urnModel AndForkTest

// AndFork followed by a PathBodyNode (Responsibility, OrJoin, AndJoin, WaitingPlace,
// Timer, FailurePoint, Connect)
// or RegularEnd (EndpointWithConnect, OrFork, AndFork, Stub)

map AndForkTestEndPoint {
  start s -> and {
    * -> ;
    * -> ;
  } -> end e.
}

map AndForkTestEndPointAgain {
  start s -> and {
    * -> ;
    * -> ;
  } -> end e.
  start s2 -> and {
    * -> ;
    * -> ;
  } -> end e.
}

map AndForkTestResponsibility {
  start s -> and {
    * -> ;
    * -> ;
  } -> X r1 -> end e.
}

map AndForkTestResponsibilityAgain {
  start s -> and {
    * -> ;
    * -> ;
  } -> X r1 -> end e.
  start s2 -> and {
    * -> ;
    * -> ;
  } -> X r1 -> end e2.
}

map AndForkTestOrFork {
  start s -> and {
    * -> ;
    * -> ;
  } -> or {
    [condition] -> ;
  } -> end e.
}
```

```

map AndForkTestOrJoin {
  start s -> and {
    * -> ;
    * -> ;
  } -> join j -> end e.
  start s2 -> j;
}

map AndForkTestAndFork {
  start s -> and {
    * -> ;
    * -> ;
  } -> and {
    * -> ;
    * -> ;
  } -> end e.
}

map AndForkTestAndJoin {
  start s -> and {
    * -> ;
    * -> ;
  } -> sync sy -> end e.
  start s2 -> sy;
}

map AndForkTestWaitingPlace {
  start s -> and {
    * -> ;
    * -> ;
  } -> wait -> end e.
}

map AndForkTestWaitingPlaceWithWaitKind {
  start s -> and {
    * -> ;
    * -> ;
  } -> transient wait -> end e.
}

map AndForkTestTimer {
  start s -> and {
    * -> ;
    * -> ;
  } -> timer {} -> end e.
}

map AndForkTestTimerWithWaitKind {
  start s -> and {
    * -> ;
    * -> ;
  } -> persistent timer {} -> end e.
}

```

```

map AndForkTestFailurePoint {
  start s -> and {
    * -> ;
    * -> ;
  } -> fail -> end e.
}

map AndForkTestStub {
  start s -> and {
    * -> ;
    * -> ;
  } -> stub () -> end e.
}

map AndForkTestStubWithStubType {
  start s -> and {
    * -> ;
    * -> ;
  } -> blocking stub () -> end e.
}

map AndForkTestAsynchronousConnectStartPoint {
  start s -> and {
    * -> ;
    * -> ;
  } -> trigger s2; -> end e.
  start s2 -> end e2.
}

map AndForkTestAsynchronousConnectWaitingPlace {
  start s -> and {
    * -> ;
    * -> ;
  } -> trigger wp; -> end e.
  start s2 -> wait wp -> end e2.
}

map AndForkTestAsynchronousConnectTimer {
  start s -> and {
    * -> ;
    * -> ;
  } -> trigger t; -> end e.
  start s2 -> timer t -> end e2.
}

// AndFork followed by a ReferencedEnd
// (Responsibility, OrJoin, AndJoin, FailurePoint, Stub)

map AndForkTestReferencedResponsibility {
  start s -> X r1 -> end e.
  start s2 -> and {
    * -> ;
    * -> ;
  } -> r1;
}

```

```

map AndForkTestReferencedOrJoin {
  start s -> join j -> end e.
  start s2 -> and {
    * -> ;
    * -> ;
  } -> j;
}

map AndForkTestReferencedAndJoin {
  start s -> sync sy -> end e.
  start s2 -> and {
    * -> ;
    * -> ;
  } -> sy;
}

map AndForkTestReferencedFailurePoint {
  start s -> fail f -> end e.
  start s2 -> and {
    * -> ;
    * -> ;
  } -> f;
}

map AndForkTestReferencedStub {
  start s -> stub st () -> end e.
  start s2 -> and {
    * -> ;
    * -> ;
  } -> in st 1;
}

map AndForkTestReferencedStubInpath2 {
  start s -> stub st () -> end e.
  start s2 -> and {
    * -> ;
    * -> ;
  } -> in st 2;
}

// branches of AndFork with all possible path nodes as first element on branch

map AndForkTestBranches {
  start s -> and {
    * -> end e.
    * -> end e1.
    * -> end e1.
    * -> X r1 -> end e2.
    * -> X r1 -> end e3.
    * -> or {
      [condition] -> end e4.
      [!condition] -> end e5.
    }
  }
  * -> join j -> end e6.
}

```

```

* -> and {
*   -> end e7.
*   -> end e8.
* }
* -> sync sy -> end e9.
* -> wait -> end e10.
* -> transient wait -> end e11.
* -> timer {} -> end e12.
* -> persistent timer {} -> end e13.
* -> fail -> end e14.
* -> stub () ; -> end e15.
* -> blocking stub () ; -> end e16.
* -> trigger s4; -> end e17.
* -> trigger wp; -> end e18.
* -> trigger t; -> end e19.
* -> r2;
* -> j2;
* -> sy2;
* -> f;
* -> in st2 1;
* -> in st2 2;
* }
start s2 -> j;
start s3 -> sy;
start s4 -> end e44.
start s5 -> wait wp -> end e55.
start s6 -> timer t -> end e66.
start s7 -> X r2 -> end e77.
start s8 -> join j2 -> end e88.
start s9 -> sync sy2 -> end e99.
start s10 -> fail f -> end e1010.
start s11 -> stub st2 () ; -> end e1111.
}

map AndForkTestMergedBranches {
  start s -> and {
    * -> ;
    * -> X r1 -> ;
    * -> X r1 -> ;
    * -> or {
      [condition] -> ;
      [!condition] -> ;
    } -> ;
    * -> join j -> ;
    * -> and {
      * -> ;
      * -> ;
    } -> ;
    * -> sync sy -> ;
    * -> wait -> ;
    * -> transient wait -> ;
    * -> timer {} -> ;
    * -> persistent timer {} -> ;
    * -> fail -> ;
  }
}

```

```

* -> stub () ; -> ;
* -> blocking stub () ; -> ;
* -> trigger s4; -> ;
* -> trigger wp; -> ;
* -> trigger t; -> ;
* -> r2;
* -> j2;
* -> sy2;
* -> f;
* -> in st2 1;
* -> in st2 2;
} -> end e.
start s2 -> j;
start s3 -> sy;
start s4 -> end e44.
start s5 -> wait wp -> end e55.
start s6 -> timer t -> end e66.
start s7 -> X r2 -> end e77.
start s8 -> join j2 -> end e88.
start s9 -> sync sy2 -> end e99.
start s10 -> fail f -> end e1010.
start s11 -> stub st2 () ; -> end e1111.
}

// full specification of AndFork
map AndForkTestFull {
  start s -> and a#"AndFork One" {
    * -> ;
    * -> ;
  } -> end e.
}

map AndForkTestNoConnectingBody {
  start s -> and a#"AndFork One" {
    * -> end e1.
    * -> end e2.
  }
}

map AndForkTestRegularBranchAndConnectingBranch {
  start s -> and a#"AndFork One" {
    * -> end e1.
    * -> ;
  } -> end e2.
}

map AndForkTestRegularBranchAndTwoConnectingBranches {
  start s -> and a#"AndFork One" {
    * -> end e1.
    * -> ;
    * -> ;
  } -> end e2.
}

```

```
// validation checks of AndFork
```

```
map AndForkTestNoBranch {  
  start s -> and a {  
    } -> end e.  
}
```

```
map AndForkTestOnlyOneBranch {  
  start s -> and a {  
    * -> ;  
  } -> end e.  
}
```

## Test cases for AndJoin

```
urnModel AndJoinTest

// AndJoin followed by a PathBodyNode (Responsibility, OrJoin, AndJoin, WaitingPlace,
// Timer, FailurePoint, Connect)
// or RegularEnd (EndpointWithConnect, OrFork, AndFork, Stub)

map AndJoinTestEndPoint {
  start s -> sync sy -> end e.
  start s2 -> sy;
}

map AndJoinTestEndPointAgain {
  start s -> sync sy -> end e.
  start s2 -> sy;
  start s3 -> sync sy2 -> end e.
  start s4 -> sy2;
}

map AndJoinTestResponsibility {
  start s -> sync sy -> X r1 -> end e.
  start s2 -> sy;
}

map AndJoinTestResponsibilityAgain {
  start s -> sync sy -> X r1 -> end e.
  start s2 -> sy;
  start s3 -> sync sy2 -> X r1 -> end e2.
  start s4 -> sy2;
}

map AndJoinTestOrFork {
  start s -> sync sy -> or {
    [condition] -> ;
  } -> end e.
  start s2 -> sy;
}

map AndJoinTestOrJoin {
  start s -> sync sy -> join j -> end e.
  start s2 -> sy;
  start s3 -> j;
}

map AndJoinTestAndFork {
  start s -> sync sy -> and {
    * -> ;
    * -> ;
  } -> end e.
  start s2 -> sy;
}

map AndJoinTestAndJoin {
  start s -> sync sy -> sync sy2 -> end e.
  start s2 -> sy;
  start s3 -> sy2;
}
```



```

map AndJoinTestWaitingPlace {
    start s -> sync sy -> wait -> end e.
    start s2 -> sy;
}

map AndJoinTestWaitingPlaceWithWaitKind {
    start s -> sync sy -> transient wait -> end e.
    start s2 -> sy;
}

map AndJoinTestTimer {
    start s -> sync sy -> timer {} -> end e.
    start s2 -> sy;
}

map AndJoinTestTimerWithWaitKind {
    start s -> sync sy -> persistent timer {} -> end e.
    start s2 -> sy;
}

map AndJoinTestFailurePoint {
    start s -> sync sy -> fail -> end e.
    start s2 -> sy;
}

map AndJoinTestStub {
    start s -> sync sy -> stub () -> end e.
    start s2 -> sy;
}

map AndJoinTestStubWithStubType {
    start s -> sync sy -> blocking stub () -> end e.
    start s2 -> sy;
}

map AndJoinTestAsynchronousConnectStartPoint {
    start s -> sync sy -> trigger s3; -> end e.
    start s2 -> sy;
    start s3 -> end e2.
}

map AndJoinTestAsynchronousConnectWaitingPlace {
    start s -> sync sy -> trigger wp; -> end e.
    start s2 -> sy;
    start s3 -> wait wp -> end e2.
}

map AndJoinTestAsynchronousConnectTimer {
    start s -> sync sy -> trigger t; -> end e.
    start s2 -> sy;
    start s3 -> timer t -> end e2.
}

```

```

// AndJoin followed by a ReferencedEnd
// (Responsibility, OrJoin, AndJoin, FailurePoint, Stub)

map AndJoinTestReferencedResponsibility {
    start s -> X r1 -> end e.
    start s2 -> sync sy -> r1;
    start s3 -> sy;
}

map AndJoinTestReferencedOrJoin {
    start s -> join j -> end e.
    start s2 -> sync sy -> j;
    start s3 -> sy;
}

map AndJoinTestReferencedAndJoin {
    start s -> sync sy2 -> end e.
    start s2 -> sync sy -> sy2;
    start s3 -> sy;
}

map AndJoinTestReferencedFailurePoint {
    start s -> fail f -> end e.
    start s2 -> sync sy -> f;
    start s3 -> sy;
}

map AndJoinTestReferencedStub {
    start s -> stub st () -> end e.
    start s2 -> sync sy -> in st 1;
    start s3 -> sy;
}

map AndJoinTestReferencedStubInpath2 {
    start s -> stub st () -> end e.
    start s2 -> sync sy -> in st 2;
    start s3 -> sy;
}

// full specification of AndJoin

map AndJoinTestFull {
    start s -> sync sy#"AndJoin One" -> end e.
    start s2 -> sy;
}

```

## Test cases for WaitingPlace

```
urnModel WaitingPlaceTest

// WaitingPlace followed by a PathBodyNode (Responsibility, OrJoin, AndJoin, WaitingPlace,
// Timer, FailurePoint, Connect)
// or RegularEnd (EndpointWithConnect, OrFork, AndFork, Stub)

map WaitingPlaceTestEndPoint {
  start s -> wait -> end e.
}

map WaitingPlaceTestEndPointAgain {
  start s -> wait -> end e.
  start s2 -> wait -> end e.
}

map WaitingPlaceTestResponsibility {
  start s -> wait -> X r1 -> end e.
}

map WaitingPlaceTestResponsibilityAgain {
  start s -> wait -> X r1 -> end e.
  start s2 -> wait -> X r1 -> end e2.
}

map WaitingPlaceTestOrFork {
  start s -> wait -> or {
    [condition] -> ;
  } -> end e.
}

map WaitingPlaceTestOrJoin {
  start s -> wait -> join j -> end e.
  start s2 -> j;
}

map WaitingPlaceTestAndFork {
  start s -> wait -> and {
    * -> ;
    * -> ;
  } -> end e.
}

map WaitingPlaceTestAndJoin {
  start s -> wait -> sync sy -> end e.
  start s2 -> sy;
}

map WaitingPlaceTestWaitingPlace {
  start s -> wait -> wait -> end e.
}

map WaitingPlaceTestWaitingPlaceWithWaitKind {
  start s -> wait -> transient wait -> end e.
}
```

```

map WaitingPlaceTestTimer {
  start s -> wait -> timer {} -> end e.
}

map WaitingPlaceTestTimerWithWaitKind {
  start s -> wait -> persistent timer {} -> end e.
}

map WaitingPlaceTestFailurePoint {
  start s -> wait -> fail -> end e.
}

map WaitingPlaceTestStub {
  start s -> wait -> stub () -> end e.
}

map WaitingPlaceTestStubWithStubType {
  start s -> wait -> blocking stub () -> end e.
}

map WaitingPlaceTestAsynchronousConnectStartPoint {
  start s -> wait -> trigger s2; -> end e.
  start s2 -> end e2.
}

map WaitingPlaceTestAsynchronousConnectWaitingPlace {
  start s -> wait -> trigger wp; -> end e.
  start s2 -> wait wp -> end e2.
}

map WaitingPlaceTestAsynchronousConnectTimer {
  start s -> wait -> trigger t; -> end e.
  start s2 -> timer t -> end e2.
}

// WaitingPlace followed by a ReferencedEnd
// (Responsibility, OrJoin, AndJoin, FailurePoint, Stub)

map WaitingPlaceTestReferencedResponsibility {
  start s -> X r1 -> end e.
  start s2 -> wait -> r1;
}

map WaitingPlaceTestReferencedOrJoin {
  start s -> join j -> end e.
  start s2 -> wait -> j;
}

map WaitingPlaceTestReferencedAndJoin {
  start s -> sync sy -> end e.
  start s2 -> wait -> sy;
}

```

```

map WaitingPlaceTestReferencedFailurePoint {
    start s -> fail f -> end e.
    start s2 -> wait -> f;
}

map WaitingPlaceTestReferencedStub {
    start s -> stub st () -> end e.
    start s2 -> wait -> in st 1;
}

map WaitingPlaceTestReferencedStubInpath2 {
    start s -> stub st () -> end e.
    start s2 -> wait -> in st 2;
}

// full specification of WaitingPlace

map WaitingPlaceTestFull {
    start s -> wait wp#"WaitingPlace One" -> end e.
}

map WaitingPlaceTestPersistent {
    start s -> persistent wait wp#"WaitingPlace One" -> end e.
}

map WaitingPlaceTestTransient {
    start s -> transient wait wp#"WaitingPlace One" -> end e.
}

map WaitingPlaceTestCondition {
    start s -> wait wp#"WaitingPlace One" [condition] -> end e.
}

map WaitingPlaceTestJustCondition {
    start s -> wait [condition] -> end e.
}

```

## Test cases for Timer

```
urnModel TimerTest

// Timer followed by a PathBodyNode (Responsibility, OrJoin, AndJoin, WaitingPlace, Timer,
// FailurePoint, Connect)
// or RegularEnd (EndpointWithConnect, OrFork, AndFork, Stub)

map TimerTestEndPoint {
  start s -> timer {} -> end e.
}

map TimerTestEndPointAgain {
  start s -> timer {} -> end e.
  start s2 -> timer {} -> end e.
}

map TimerTestResponsibility {
  start s -> timer {} -> X r1 -> end e.
}

map TimerTestResponsibilityAgain {
  start s -> timer {} -> X r1 -> end e.
  start s2 -> timer {} -> X r1 -> end e2.
}

map TimerTestOrFork {
  start s -> timer {} -> or {
    [condition] -> ;
  } -> end e.
}

map TimerTestOrJoin {
  start s -> timer {} -> join j -> end e.
  start s2 -> j;
}

map TimerTestAndFork {
  start s -> timer {} -> and {
    * -> ;
    * -> ;
  } -> end e.
}

map TimerTestAndJoin {
  start s -> timer {} -> sync sy -> end e.
  start s2 -> sy;
}

map TimerTestWaitingPlace {
  start s -> timer {} -> wait -> end e.
}

map TimerTestWaitingPlaceWithWaitKind {
  start s -> timer {} -> transient wait -> end e.
}
```

```

map TimerTestTimer {
  start s -> timer {} -> timer {} -> end e.
}

map TimerTestTimerWithWaitKind {
  start s -> timer {} -> persistent timer {} -> end e.
}

map TimerTestFailurePoint {
  start s -> timer {} -> fail -> end e.
}

map TimerTestStub {
  start s -> timer {} -> stub () -> end e.
}

map TimerTestStubWithStubType {
  start s -> timer {} -> blocking stub () -> end e.
}

map TimerTestAsynchronousConnectStartPoint {
  start s -> timer {} -> trigger s2; -> end e.
  start s2 -> end e2.
}

map TimerTestAsynchronousConnectWaitingPlace {
  start s -> timer {} -> trigger wp; -> end e.
  start s2 -> wait wp -> end e2.
}

map TimerTestAsynchronousConnectTimer {
  start s -> timer {} -> trigger t; -> end e.
  start s2 -> timer t -> end e2.
}

// Timer followed by a ReferencedEnd
// (Responsibility, OrJoin, AndJoin, FailurePoint, Stub)

map TimerTestReferencedResponsibility {
  start s -> X r1 -> end e.
  start s2 -> timer {} -> r1;
}

map TimerTestReferencedOrJoin {
  start s -> join j -> end e.
  start s2 -> timer {} -> j;
}

map TimerTestReferencedAndJoin {
  start s -> sync sy -> end e.
  start s2 -> timer {} -> sy;
}

```

```

map TimerTestReferencedFailurePoint {
  start s -> fail f -> end e.
  start s2 -> timer {} -> f;
}

map TimerTestReferencedStub {
  start s -> stub st () -> end e.
  start s2 -> timer {} -> in st 1;
}

map TimerTestReferencedStubInpath2 {
  start s -> stub st () -> end e.
  start s2 -> timer {} -> in st 2;
}

// full specification of Timer

map TimerTestFull {
  start s -> timer t#"Timer One" {} -> end e.
}

map TimerTestPersistent {
  start s -> persistent timer t#"Timer One" {} -> end e.
}

map TimerTestTransient {
  start s -> transient timer t#"Timer One" {} -> end e.
}

map TimerTestCondition {
  start s -> timer t#"Timer One" {} [condition] -> end e.
}

map TimerTestJustCondition {
  start s -> timer {} [condition] -> end e.
}

map TimerTestTimeout {
  start s -> timer t#"Timer One" { timeout -> end e2. } -> end e.
}

map TimerTestTimeoutWithCondition {
  start s -> timer t#"Timer One" { timeout [condition] -> end e2. } -> end e.
}

map TimerTestWithoutTimeoutCurlyBrackets {
  start s -> timer -> end e.
}

```



## Test cases for FailurePoint

```
urnModel FailurePointTest

// FailurePoint followed by a PathBodyNode (Responsibility, OrJoin, AndJoin, WaitingPlace,
// Timer, FailurePoint, Connect)
// or RegularEnd (EndpointWithConnect, OrFork, AndFork, Stub)

map FailurePointTestEndPoint {
  start s -> fail -> end e.
}

map FailurePointTestEndPointAgain {
  start s -> fail -> end e.
  start s2 -> fail -> end e.
}

map FailurePointTestResponsibility {
  start s -> fail -> X r1 -> end e.
}

map FailurePointTestResponsibilityAgain {
  start s -> fail -> X r1 -> end e.
  start s2 -> fail -> X r1 -> end e2.
}

map FailurePointTestOrFork {
  start s -> fail -> or {
    [condition] -> ;
  } -> end e.
}

map FailurePointTestOrJoin {
  start s -> fail -> join j -> end e.
  start s2 -> j;
}

map FailurePointTestAndFork {
  start s -> fail -> and {
    * -> ;
    * -> ;
  } -> end e.
}

map FailurePointTestAndJoin {
  start s -> fail -> sync sy -> end e.
  start s2 -> sy;
}

map FailurePointTestWaitingPlace {
  start s -> fail -> wait -> end e.
}

map FailurePointTestWaitingPlaceWithWaitKind {
  start s -> fail -> transient wait -> end e.
}
```

```

map FailurePointTestTimer {
    start s -> fail -> timer {} -> end e.
}

map FailurePointTestTimerWithWaitKind {
    start s -> fail -> persistent timer {} -> end e.
}

map FailurePointTestFailurePoint {
    start s -> fail -> fail -> end e.
}

map FailurePointTestStub {
    start s -> fail -> stub () -> end e.
}

map FailurePointTestStubWithStubType {
    start s -> fail -> blocking stub () -> end e.
}

map FailurePointTestAsynchronousConnectStartPoint {
    start s -> fail -> trigger s2; -> end e.
    start s2 -> end e2.
}

map FailurePointTestAsynchronousConnectWaitingPlace {
    start s -> fail -> trigger wp; -> end e.
    start s2 -> wait wp -> end e2.
}

map FailurePointTestAsynchronousConnectTimer {
    start s -> fail -> trigger t; -> end e.
    start s2 -> timer t -> end e2.
}

// FailurePoint followed by a ReferencedEnd
// (Responsibility, OrJoin, AndJoin, FailurePoint, Stub)

map FailurePointTestReferencedResponsibility {
    start s -> X r1 -> end e.
    start s2 -> fail -> r1;
}

map FailurePointTestReferencedOrJoin {
    start s -> join j -> end e.
    start s2 -> fail -> j;
}

map FailurePointTestReferencedAndJoin {
    start s -> sync sy -> end e.
    start s2 -> fail -> sy;
}

```

```

map FailurePointTestReferencedFailurePoint {
  start s -> fail f -> end e.
  start s2 -> fail -> f;
}

map FailurePointTestReferencedStub {
  start s -> stub st () -> end e.
  start s2 -> fail -> in st 1;
}

map FailurePointTestReferencedStubInpath2 {
  start s -> stub st () -> end e.
  start s2 -> fail -> in st 2;
}

// full specification of FailurePoint

map FailurePointTestFull {
  start s -> fail f#"FailurePoint One" -> end e.
}

map FailurePointTestWithException1 {
  start s -> fail f#"FailurePoint One" [condition] ["exception"] -> end e.
}

map FailurePointTestWithException2 {
  start s -> fail f#"FailurePoint One" ["exception"] [condition] -> end e.
}

map FailurePointTestJustWithException1 {
  start s -> fail [condition] ["exception"] -> end e.
}

map FailurePointTestJustWithException2 {
  start s -> fail ["exception"] [condition] -> end e.
}

```

## Test cases for Stub

```
urnModel StubTest

// Stub followed by a PathBodyNode (Responsibility, OrJoin, AndJoin, WaitingPlace, Timer,
// FailurePoint, Connect)
// or RegularEnd (EndpointWithConnect, OrFork, AndFork, Stub)

map StubTestEndPoint {
  start s -> stub () -> end e.
}

map StubTestEndPointAgain {
  start s -> stub () -> end e.
  start s2 -> stub () -> end e.
}

map StubTestResponsibility {
  start s -> stub () -> X r1 -> end e.
}

map StubTestResponsibilityAgain {
  start s -> stub () -> X r1 -> end e.
  start s2 -> stub () -> X r1 -> end e2.
}

map StubTestOrFork {
  start s -> stub () -> or {
    [condition] -> ;
  } -> end e.
}

map StubTestOrJoin {
  start s -> stub () -> join j -> end e.
  start s2 -> j;
}

map StubTestAndFork {
  start s -> stub () -> and {
    * -> ;
    * -> ;
  } -> end e.
}

map StubTestAndJoin {
  start s -> stub () -> sync sy -> end e.
  start s2 -> sy;
}

map StubTestWaitingPlace {
  start s -> stub () -> wait -> end e.
}

map StubTestWaitingPlaceWithWaitKind {
  start s -> stub () -> transient wait -> end e.
}
```

```

map StubTestTimer {
    start s -> stub () -> timer {} -> end e.
}

map StubTestTimerWithWaitKind {
    start s -> stub () -> persistent timer {} -> end e.
}

map StubTestFailurePoint {
    start s -> stub () -> fail -> end e.
}

map StubTestStub {
    start s -> stub () -> stub () -> end e.
}

map StubTestStubWithStubType {
    start s -> stub () -> blocking stub () -> end e.
}

map StubTestAsynchronousConnectStartPoint {
    start s -> stub () -> trigger s2; -> end e.
    start s2 -> end e2.
}

map StubTestAsynchronousConnectWaitingPlace {
    start s -> stub () -> trigger wp; -> end e.
    start s2 -> wait wp -> end e2.
}

map StubTestAsynchronousConnectTimer {
    start s -> stub () -> trigger t; -> end e.
    start s2 -> timer t -> end e2.
}

// Stub followed by a ReferencedEnd
// (Responsibility, OrJoin, AndJoin, FailurePoint, Stub)

map StubTestReferencedResponsibility {
    start s -> X r1 -> end e.
    start s2 -> stub () -> r1;
}

map StubTestReferencedOrJoin {
    start s -> join j -> end e.
    start s2 -> stub () -> j;
}

map StubTestReferencedAndJoin {
    start s -> sync sy -> end e.
    start s2 -> stub () -> sy;
}

```

```

map StubTestReferencedFailurePoint {
  start s -> fail f -> end e.
  start s2 -> stub () -> f;
}

map StubTestReferencedStub {
  start s -> stub st ()-> end e.
  start s2 -> stub () -> in st 1;
}

map StubTestReferencedStubInpath2 {
  start s -> stub st ()-> end e.
  start s2 -> stub () -> in st 2;
}

// full specification of Stub

map StubTestFull {
  start s -> stub s#"Stub One" () -> end e.
}

map StubTestSynchronizing {
  start s -> synchronizing stub s#"Stub One" () -> end e.
}

map StubTestBlocking {
  start s -> blocking stub s#"Stub One" () -> end e.
}

map StubTestOneConnectingOutpath {
  start s -> stub (
    PluginOne: sOne1=in 1, eOne1=out 1
  ) {
    out 1 -> ;
  } -> end e.
}

map StubTestTwoConnectingOutpaths {
  start s -> stub (
    PluginOne: sOne1=in 1, eOne1=out 1, eOne2=out 2
  ) {
    out 1 -> ;
    out 2 -> ;
  } -> end e.
}

map StubTestNoConnectingOutpaths {
  start s -> stub (
    PluginOne: sOne1=in 1, eOne1=out 1, eOne2=out 2
  ) {
    out 1 -> end e1.
    out 2 -> end e2.
  }
}

```

```

map StubTestRegularOutpathAndConnectingOutpath {
  start s -> stub (
    PluginOne: sOne1=in 1, eOne1=out 1, eOne2=out 2
  ) {
    out 1 -> end e1.
    out 2 -> ;
  } -> end e2.
}

map StubTestRegularOutpathTwoConnectingOutpaths {
  start s -> stub (
    PluginOne: sOne1=in 1, eOne1=out 1, eOne2=out 2, eOne3=out 3
  ) {
    out 1 -> end e1.
    out 2 -> ;
    out 3 -> ;
  } -> end e2.
}

map StubTestTwoPlugins {
  start s -> stub (
    PluginOne: sOne1=in 1, eOne1=out 1
    PluginTwo: sTwo=in 1, eTwo=out 1
  ) {
    out 1 -> ;
  } -> end e.
}

map StubTestMultipleInAndOutPaths {
  start s -> stub st (
    PluginThree: sThree1=in 1, sThree2=in 2, eThree1=out 1, eThree2=out 2, eThree3=out 3
  ) {
    out 1 -> end e1.
    out 2 -> end e2.
    out 3 -> end e3.
  }
  start s2 -> in st 2;
}

map StubTestOutpathWithThreshold {
  start s -> synchronizing stub (
    PluginOne: sOne1=in 1, eOne1=out 1
  ) {
    out 1 threshold [2] -> ;
  } -> end e.
}

map StubTestPluginWithConditionAndReplication {
  start s -> stub (
    [condition] replication 2 PluginOne: sOne1=in 1, eOne1=out 1
  ) {
    out 1 -> ;
  } -> end e.
}

```

```

map StubTestStubDeclaration {
  st = (
    PluginOne: sOne1=in 1, eOne1=out 1
  )
  start s -> stub st {
    out 1 -> ;
  } -> end e.
}

map PluginOne {
  start sOne1 -> or {
    [condition1] -> end eOne1.
    [condition2] -> end eOne2.
    [condition3] -> end eOne3.
  }
}

map PluginTwo {
  start sTwo -> end eTwo.
}

map PluginThree {
  start sThree1 -> or {
    [condition1] -> end eThree1.
    [condition2] -> join j -> end eThree2.
  }
  start sThree2 -> and {
    * -> j;
    * -> end eThree3.
  }
}

```



## Test cases for Connect

```
urnModel1 ConnectTest

// Connect followed by a PathBodyNode (Responsibility, OrJoin, AndJoin, WaitingPlace,
// Timer, FailurePoint, Connect)
// or RegularEnd (EndpointWithConnect, OrFork, AndFork, Stub)
// (only WaitingPlace is triggered by Connect; not tested: StartPoint and Timer)

map ConnectTestEndPoint {
  start s -> trigger wp; -> end e.
  start s2 -> wait wp -> end e2.
}

map ConnectTestEndPointAgain {
  start s -> trigger wp; -> end e.
  start s2 -> wait wp -> end e2.
  start s3 -> trigger wp2; -> end e.
  start s4 -> wait wp2 -> end e4.
}

map ConnectTestResponsibility {
  start s -> trigger wp; -> X r1 -> end e.
  start s2 -> wait wp -> end e2.
}

map ConnectTestResponsibilityAgain {
  start s -> trigger wp; -> X r1 -> end e.
  start s2 -> wait wp -> end e2.
  start s3 -> trigger wp2; -> X r1 -> end e3.
  start s4 -> wait wp2 -> end e4.
}

map ConnectTestOrFork {
  start s -> trigger wp; -> or {
    [condition] -> ;
  } -> end e.
  start s2 -> wait wp -> end e2.
}

map ConnectTestOrJoin {
  start s -> trigger wp; -> join j -> end e.
  start s2 -> wait wp -> end e2.
  start s3 -> j;
}

map ConnectTestAndFork {
  start s -> trigger wp; -> and {
    * -> ;
    * -> ;
  } -> end e.
  start s2 -> wait wp -> end e2.
}
```

```

map ConnectTestAndJoin {
  start s -> trigger wp; -> sync sy -> end e.
  start s2 -> wait wp -> end e2.
  start s3 -> sy;
}

map ConnectTestWaitingPlace {
  start s -> trigger wp; -> wait -> end e.
  start s2 -> wait wp -> end e2.
}

map ConnectTestWaitingPlaceWithWaitKind {
  start s -> trigger wp; -> transient wait -> end e.
  start s2 -> wait wp -> end e2.
}

map ConnectTestTimer {
  start s -> trigger wp; -> timer {} -> end e.
  start s2 -> wait wp -> end e2.
}

map ConnectTestTimerWithWaitKind {
  start s -> trigger wp; -> persistent timer {} -> end e.
  start s2 -> wait wp -> end e2.
}

map ConnectTestFailurePoint {
  start s -> trigger wp; -> fail -> end e.
  start s2 -> wait wp -> end e2.
}

map ConnectTestStub {
  start s -> trigger wp; -> stub () -> end e.
  start s2 -> wait wp -> end e2.
}

map ConnectTestStubWithStubType {
  start s -> trigger wp; -> blocking stub () -> end e.
  start s2 -> wait wp -> end e2.
}

map ConnectTestAsynchronousConnectStartPoint {
  start s -> trigger wp; -> trigger s3; -> end e.
  start s2 -> wait wp -> end e2.
  start s3 -> end e3.
}

map ConnectTestAsynchronousConnectWaitingPlace {
  start s -> trigger wp; -> trigger wp2; -> end e.
  start s2 -> wait wp -> end e2.
  start s3 -> wait wp2 -> end e3.
}

```

```

map ConnectTestAsynchronousConnectTimer {
  start s -> trigger wp; -> trigger t; -> end e.
  start s2 -> wait wp -> end e2.
  start s3 -> timer t -> end e3.
}

// Connect followed by a ReferencedEnd
// (Responsibility, OrJoin, AndJoin, FailurePoint, Stub)

map ConnectTestReferencedResponsibility {
  start s -> X r1 -> end e.
  start s2 -> trigger wp; -> r1;
  start s3 -> wait wp -> end e3.
}

map ConnectTestReferencedOrJoin {
  start s -> join j -> end e.
  start s2 -> trigger wp; -> j;
  start s3 -> wait wp -> end e3.
}

map ConnectTestReferencedAndJoin {
  start s -> sync sy -> end e.
  start s2 -> trigger wp; -> sy;
  start s3 -> wait wp -> end e3.
}

map ConnectTestReferencedFailurePoint {
  start s -> fail f -> end e.
  start s2 -> trigger wp; -> f;
  start s3 -> wait wp -> end e3.
}

map ConnectTestReferencedStub {
  start s -> stub st () -> end e.
  start s2 -> trigger wp; -> in st 1;
  start s3 -> wait wp -> end e3.
}

map ConnectTestReferencedStubInpath2 {
  start s -> stub st () -> end e.
  start s2 -> trigger wp; -> in st 2;
  start s3 -> wait wp -> end e3.
}

// full specification of Connect
// (nothing to test)

```

## Test cases for Component

```
urnModel ComponentTest

map ComponentTestSingleComponent {
  start s -> X r1 -> end e.
  team C: s, r1, e
}

map ComponentTestTwoComponents {
  start s -> X r1 -> end e.
  team C1: s, e
  team C2: r1
}

map ComponentTestNestedComponents {
  start s -> X r1 -> end e.
  team C1: s, e, C2
  team C2: r1
}

map ComponentTestFromToSingleComponent {
  start s -> X r1 -> or {
    [condition] -> X r2 -> end e.
    [!condition] -> X r3 -> end e2.
  }
  team C: s..e
}

map ComponentTestFromToTwoComponents {
  start s -> X r1 -> or {
    [condition] -> X r2 -> end e.
    [!condition] -> X r3 -> end e2.
  }
  team C1: s..e
  team C2: r3..e2
}

map ComponentTestFromToSecondComponentInMiddle {
  start s -> X r1 -> or {
    [condition] -> X r2 -> end e.
    [!condition] -> X r3 -> end e2.
  }
  team C1: s..e
  team C2: r2
}

map ComponentTestProtectedComponent {
  start s -> X r1 -> end e.
  protected team Cp: s, r1, e
}

map ComponentTestActor {
  start s -> X r1 -> end e.
  actor A: s..e
}
```

```

map ComponentTestParentComponent {
  start s -> X r1 -> end e.
  parent C: s..e
}

map ParentMap {
  start s -> stub st (
    ComponentTestParentComponent: s=in 1, e=out 1, C=CParent
  ) {
    out 1 -> end e.
  }
  team CParent: s..e
}

map ComponentTestLongNames {
  start s -> X rA -> X rB_1#"Beta" -> end e.
  start s2 -> X rA -> X rB_2#"Beta" -> end e2.
  team C1: rA, rB_1
  team C2: rB_2
}

```

## Test cases for Softgoal

```
urnModel SoftgoalTest

// Softgoal followed by an ElementLink (Contribution, Decomposition, Dependency) to
// another IntentionalElement (Softgoal, Goal, Task in case of Contribution;
// Softgoal, Goal, Task, Resource in case of Decomposition;
// Softgoal, Goal, Task, Resource, Indicator in case of Dependency)

actor SG#"Softgoal Test for ElementLinks" {
  softgoal sgSource {
    contributesTo sg1 with make
    contributesTo g1 with 100
    contributesTo t1 correlated with break
    and decomposes sg2
    or decomposes g2
    xor decomposes t2
    and decomposes r2
    dependsOn Other.sg3
    dependsOn Other.g3
    dependsOn Other.t3
    dependsOn Other.r3
    dependsOn Other.i3
  }
  softgoal sg1 {}
  softgoal sg2 {}
  goal g1 {}
  goal g2 {}
  task t1{}
  task t2 {}
  resource r2 {}
}

actor Other {
  softgoal sg3 {}
  goal g3 {}
  task t3 {}
  resource r3 {}
  indicator i3 {}
}

// full specification of Softgoal

actor SG2#"Softgoal Test for Optional Elements" {
  softgoal sgSource2#"Softgoal" {
    importance 100
  }
}
```

## Test cases for Goal

```
urnModel GoalTest

// Goal followed by an ElementLink (Contribution, Decomposition, Dependency) to
// another IntentionalElement (Softgoal, Goal, Task in case of Contribution;
// Softgoal, Goal, Task, Resource in case of Decomposition;
// Softgoal, Goal, Task, Resource, Indicator in case of Dependency)

actor G#"Goal Test for ElementLinks" {
  goal gSource {
    contributesTo sg1 with make
    contributesTo g1 with 100
    contributesTo t1 correlated with break
    and decomposes sg2
    or decomposes g2
    xor decomposes t2
    and decomposes r2
    dependsOn Other.sg3
    dependsOn Other.g3
    dependsOn Other.t3
    dependsOn Other.r3
    dependsOn Other.i3
  }
  softgoal sg1 {}
  softgoal sg2 {}
  goal g1 {}
  goal g2 {}
  task t1{}
  task t2 {}
  resource r2 {}
}

actor Other {
  softgoal sg3 {}
  goal g3 {}
  task t3 {}
  resource r3 {}
  indicator i3 {}
}

// full specification of Goal

actor G2#"Goal Test for Optional Elements" {
  goal gSource2#"Goal" {
    importance 100
  }
}
```

## Test cases for Task

```
urnModel TaskTest

// Task followed by an ElementLink (Contribution, Decomposition, Dependency) to
// another IntentionalElement (Softgoal, Goal, Task in case of Contribution;
// Softgoal, Goal, Task, Resource in case of Decomposition;
// Softgoal, Goal, Task, Resource, Indicator in case of Dependency)

actor T#"Task Test for ElementLinks" {
  task tSource {
    contributesTo sg1 with make
    contributesTo g1 with 100
    contributesTo t1 correlated with break
    and decomposes sg2
    or decomposes g2
    xor decomposes t2
    and decomposes r2
    dependsOn Other.sg3
    dependsOn Other.g3
    dependsOn Other.t3
    dependsOn Other.r3
    dependsOn Other.i3
  }
  softgoal sg1 {}
  softgoal sg2 {}
  goal g1 {}
  goal g2 {}
  task t1{}
  task t2 {}
  resource r2 {}
}

actor Other {
  softgoal sg3 {}
  goal g3 {}
  task t3 {}
  resource r3 {}
  indicator i3 {}
}

// full specification of Task

actor T2#"Task Test for Optional Elements" {
  task tSource2#"Task" {
    importance 100
  }
}
```



## Test cases for Resource

```
urnModel ResourceTest

// Resource followed by an ElementLink (Contribution, Decomposition, Dependency) to
// another IntentionalElement (Softgoal, Goal, Task in case of Contribution;
// Softgoal, Goal, Task, Resource in case of Decomposition;
// Softgoal, Goal, Task, Resource, Indicator in case of Dependency)

actor R#"Resource Test for ElementLinks" {
  resource rSource {
    contributesTo sg1 with make
    contributesTo g1 with 100
    contributesTo t1 correlated with break
    and decomposes sg2
    or decomposes g2
    xor decomposes t2
    and decomposes r2
    dependsOn Other.sg3
    dependsOn Other.g3
    dependsOn Other.t3
    dependsOn Other.r3
    dependsOn Other.i3
  }
  softgoal sg1 {}
  softgoal sg2 {}
  goal g1 {}
  goal g2 {}
  task t1{}
  task t2 {}
  resource r2 {}
}

actor Other {
  softgoal sg3 {}
  goal g3 {}
  task t3 {}
  resource r3 {}
  indicator i3 {}
}

// full specification of Resource

actor R2#"Resource Test for Optional Elements" {
  resource rSource2#"Resource" {
    importance 100
  }
}
```

## Test cases for Indicator

```
urnModel IndicatorTest

// Indicator followed by an ElementLink (Contribution, Decomposition, Dependency) to
// another IntentionalElement (Softgoal, Goal, Task in case of Contribution;
// Softgoal, Goal, Task, Resource in case of Decomposition;
// Softgoal, Goal, Task, Resource, Indicator in case of Dependency)

actor I#"Indicator Test for ElementLinks" {
  indicator iSource {
    contributesTo sg1 with make
    contributesTo g1 with 100
    contributesTo t1 correlated with break
    and decomposes sg2
    or decomposes g2
    xor decomposes t2
    and decomposes r2
    dependsOn Other.sg3
    dependsOn Other.g3
    dependsOn Other.t3
    dependsOn Other.r3
    dependsOn Other.i3
  }
  softgoal sg1 {}
  softgoal sg2 {}
  goal g1 {}
  goal g2 {}
  task t1{}
  task t2 {}
  resource r2 {}
}

actor Other {
  softgoal sg3 {}
  goal g3 {}
  task t3 {}
  resource r3 {}
  indicator i3 {}
}

// full specification of Indicator

actor I2#"Indicator Test for Optional Elements" {
  indicator iSource2#"Indicator" {
    importance 100
  }
}
```

## Test cases for Belief

```
urnModel BeliefTest

// Belief followed by an ElementLink (only Contribution) to
// another IntentionalElement (Softgoal, Goal, Task)

actor B#"Belief Test for ElementLinks" {
  belief bSource {
    contributesTo sg1 with make
    contributesTo g1 with 100
    contributesTo t1 correlated with break
  }
  softgoal sg1 {}
  goal g1 {}
  task t1{}
}

// full specification of Belief

actor B2#"Belief Test for Optional Elements" {
  belief bSource2#"Belief" {
    importance 100
  }
}
```

## Test cases for Validation Rules (Unique Names)

```
urnModel UniqueNamesTest

map UniqueMapName {}
map UniqueMapName {}

concern UniqueConcernName: UniqueMapName
concern UniqueConcernName: UniqueMapName

actor UniqueActorName {
  goal UniqueIntentionalElementName {
    UniqueContributionName contributesTo sg1 with make
    UniqueContributionName contributesTo g1 with 100
    UniqueDecompositionName and decomposes sg2
    UniqueDecompositionName or decomposes g2
    UniqueDependencyName dependsOn Other.sg3
    UniqueDependencyName dependsOn Other.g3
  }
  softgoal UniqueIntentionalElementName {}
  softgoal sg1 {}
  goal g1 {}
  softgoal sg2 {}
  goal g2 {}
}
actor UniqueActorName {}
actor Other {
  softgoal sg3 {}
  goal g3 {}
}

strategiesGroup UniqueStrategyGroupName: s1
strategiesGroup UniqueStrategyGroupName: s2
strategy UniqueStrategyName {}
strategy UniqueStrategyName {}
strategy s1 {}
strategy s2 {}

linearConversion UniqueLinearConversionName {
  unit "failures/week/10000 connections"
  target 0
  threshold 500
  worst 10000
}
linearConversion UniqueLinearConversionName {
  unit "failures/week/20000 connections"
  target 0
  threshold 500
  worst 10000
}

mappingConversion UniqueMappingConversionName {
  unit "equipment class 1"
  real "Class 1" --> 100
}
```

```
mappingConversion UniqueMappingConversionName {  
    unit "equipment class 2"  
    real "Class 2" --> 100  
}  
  
contributionContextGroup UniqueCCGroupName: c1  
contributionContextGroup UniqueCCGroupName: c2  
contributionContext UniqueCCName {}  
contributionContext UniqueCCName {}  
contributionContext c1 {}  
contributionContext c2 {}
```

## Test cases for Validation Rules (Unique LongNames)

```
urnModel UniqueNamesTest

map m1#"UniqueMapName" {}
map m2#"UniqueMapName" {}

concern co1#"UniqueConcernName": m1
concern co2#"UniqueConcernName": m2

actor a1#"UniqueActorName" {
  goal g#"UniqueIntentionalElementName" {
    cont1#"UniqueContributionName" contributesTo sg1 with make
    cont2#"UniqueContributionName" contributesTo g1 with 100
    deco1#"UniqueDecompositionName" and decomposes sg2
    deco2#"UniqueDecompositionName" or decomposes g2
    depe1#"UniqueDependencyName" dependsOn Other.sg3
    depe2#"UniqueDependencyName" dependsOn Other.g3
  }
  softgoal sg#"UniqueIntentionalElementName" {}
  softgoal sg1 {}
  goal g1 {}
  softgoal sg2 {}
  goal g2 {}
}
actor a2#"UniqueActorName" {}
actor Other {
  softgoal sg3 {}
  goal g3 {}
}

strategiesGroup stg1#"UniqueStrategyGroupName": s1
strategiesGroup stg2#"UniqueStrategyGroupName": s2
strategy st1#"UniqueStrategyName" {}
strategy st2#"UniqueStrategyName" {}
strategy s1 {}
strategy s2 {}

linearConversion lc1#"UniqueLinearConversionName" {
  unit "failures/week/10000 connections"
  target 0
  threshold 500
  worst 10000
}
linearConversion lc2#"UniqueLinearConversionName" {
  unit "failures/week/20000 connections"
  target 0
  threshold 500
  worst 10000
}

mappingConversion mc1#"UniqueMappingConversionName" {
  unit "equipment class 1"
  real "Class 1" --> 100
}
```

```
mappingConversion mc2#"UniqueMappingConversionName" {  
  unit "equipment class 2"  
  real "Class 2" --> 100  
}  
  
contributionContextGroup ccg1#"UniqueCCGroupName": c1  
contributionContextGroup ccg2#"UniqueCCGroupName": c2  
contributionContext cc1#"UniqueCCName" {}  
contributionContext cc2#"UniqueCCName" {}  
contributionContext c1 {}  
contributionContext c2 {}
```

## Test cases for Remaining URN Concepts

```

urnModel Example {
    description "This is an example."
    author "Ruchika Kumar"
    created "Oct-15-2016"
    modified "Oct-15-2016"
    version "1.0"
    urnVersion "29.0"
}

concern TCConcern : TelP, TCS, TL

link TelP --> OriginatingAgent
link TelP --> TerminatingAgent

metadata TCS: popularity = 90
metadata TL: popularity = 35

actor TelP#"Telecom Provider" {
    importance 100
    goal VoiceConn#"Voice Connection Be Setup" {
        importance 50
    }

    softgoal HighRel#"High Reliability" {
        importance 75
    }

    softgoal SpecUsage#"Minimize Spectrum Usage" {
        importance 60
    }

    task MakeVoiceOverInternet#"Make Voice Connection Over Internet" {
        contributesTo HighRel with somePositive
        contributesTo SpecUsage correlated with somePositive
        xor decomposes VoiceConn
    }

    task MakeVoiceOverWireless#"Make Voice Connection Over Wireless" {
        contWirelessVoiceConnToHighRel contributesTo HighRel with make
        contributesTo SpecUsage correlated with someNegative
        xor decomposes VoiceConn
    }

    indicator VoiceConnFailureRate#"Failure Rate for Voice Connection Over
    Internet" {
        unit "failures/week/10000 connections"
        contVoiceConnFailureRateToInternetVoiceConn contributesTo
        MakeVoiceOverInternet with 100
        dependsOn Tech.LoggEquip
    }

    belief WirelessReliability#"Wireless is less reliable than Internet" {
        contributesTo HighRel with someNegative
    }
}

```



```

actor Tech#"Technician" {
    resource LoggEquip#"Logging Equipment" {
        dependsOn EquipSetup
    }

    task EquipSetup#"Correctly setup logging equipment" {
        importance 100
    }
}

strategiesGroup SG1: Eval1

strategy Eval1#"Internet Connection" {
    author "ITU-T"
    Tech.EquipSetup evaluation 100
    TelP.VoiceConnFailureRate real 265 convertedWith LC1
}

linearConversion LC1#"Weakly Failures" {
    unit "failures/week/10000 connections"
    target 0
    threshold 500
    worst 10000
}

mappingConversion MC1#"EquipmentClassification" {
    unit "equipment class"
    real "Class 1" --> 100
    real "Class 2" --> 47
    real "Class 3" --> -25
}

contributionContextGroup CCG1: CCP, CCPI

contributionContext CCP#"Pessimistic" {
    contWirelessVoiceConnToHighRel with help
}

contributionContext CCPI#"PessimisticIneffective" {
    contVoiceConnFailureRateToInternetVoiceConn with 50
    includes CCP
}

```

```

map SC#"Simple Connection" {
  start request -> stub Originating(
    OF: success=out 1, failPoint=out 2, startPoint=in 1, Agent=OriginatingAgent
  ) {
    out 1 -> stub Terminating(
      TF: success=out 1, reportSuccess=out 2, busy=out 3,
      failPoint=out 4, startPoint=in 1, Agent=TerminatingAgent
    ) {
      out 1 -> end ring.
      out 2 -> X forwardSignal -> end ringing.
      out 3 -> X forwardSignal -> end busy.
      out 4 -> end notify.
    }
    out 2 -> end notify.
  }
  actor OriginatingUser: request, notify, busy, ringing
  team OriginatingAgent: Originating, forwardSignal
  team TerminatingAgent: Terminating
  team TerminatingUser: ring
}

map OF #"Originating Features" {
  start startPoint -> stub OrigFeatures(
    [!subTL] Default: startPoint=in 1, continue=out 1, Agent=Agent
    [subTL] TL: startPoint=in 1, success=out 1, failPoint=out 2,
    Agent=Agent
  ) {
    out 1 -> X sendRequest -> end success.
    out 2 -> end failPoint.
  }
  parent Agent: startPoint..success, failPoint
}

map TL {
  start startPoint -> X checkTime -> or {
    [!TLactive] -> end success.
    [TLactive] -> timer getPIN {-> deny;} -> or {
      [PINvalid] -> end success.
      [!PINvalid] -> ;
    } -> X deny -> end failPoint.
  }
  start enterPIN -> end e. -> trigger getPIN ;
  parent Agent: startPoint..failPoint, success
  actor OriginatingUser: enterPIN
}

map Default {
  start startPoint -> end continue.
  parent Agent: startPoint, continue
}

```

```

map TF #"Terminating Features" {
  start startPoint -> stub TermFeatures(
    [!subTCS] Default: startPoint=in 1, continue=out 1, Agent=Agent
    [subTCS] TCS: startPoint=in 1, success=out 1, failPoint=out 2,
    Agent=Agent
  ) {
    out 1 -> or {
      [!busy] -> and {
        * -> X ringTreatment -> end success.
        * -> X ringingTreatment -> end reportSuccess.
      }
      [busy] -> X busyTreatment -> end busy.
    }
    out 2 -> end failPoint.
  }
  parent Agent: startPoint..success, startPoint..reportSuccess, startPoint..busy,
  failPoint
}

map TCS #"Terminating Call Screening (TCS)" {
  start startPoint -> X checkTCS -> or {
    [!onTCSlist] -> end success.
    [onTCSlist] -> end failPoint.
  }
  parent Agent: startPoint..success, failPoint, TCSCreeningList
  team TCSCreeningList: checkTCS
}

```