International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.111
(11/2008)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

Formal description techniques (FDT) – Application of formal description techniques

---

Notations and guidelines for the definition of ITU-T languages

Recommendation ITU-T Z.111

ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

FORMAL DESCRIPTION TECHNIQUES (FDT)	
Specification and Description Language (SDL)	Z.100–Z.109
Application of formal description techniques	**Z.110–Z.119**
Message Sequence Chart (MSC)	Z.120–Z.129
Extended Object Definition Language (eODL)	Z.130–Z.139
User Requirements Notation (URN)	Z.150–Z.159
Testing and Test Control Notation (TTCN)	Z.160–Z.179
PROGRAMMING LANGUAGES	
CHILL: The ITU-T high level language	Z.200–Z.209
MAN-MACHINE LANGUAGE	
General principles	Z.300–Z.309
Basic syntax and dialogue procedures	Z.310–Z.319
Extended MML for visual display terminals	Z.320–Z.329
Specification of the man-machine interface	Z.330–Z.349
Data-oriented human-machine interfaces	Z.350–Z.359
Human-machine interfaces for the management of telecommunications networks	Z.360–Z.379
QUALITY	
Quality of telecommunication software	Z.400–Z.409
Quality aspects of protocol-related Recommendations	Z.450–Z.459
METHODS	
Methods for validation and testing	Z.500–Z.519
MIDDLEWARE	
Processing environment architectures	Z.600–Z.609

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

Recommendation ITU-T Z.111

Notations and guidelines for the definition of ITU-T languages

Summary

Recommendation ITU-T Z.111 provides meta-grammars for ITU-T Recommendations that define ITU-T languages in the X.680 series and the Z series of ITU-T Recommendations on languages for specification, implementation, modelling and testing. This allows the description of these meta-grammars that define the abstract or concrete grammar (syntax, constraints and semantics) of languages without having to repeat the meta-grammar (such as lexical naming rules, or the description of Backus-Naur Form syntax) as a preamble or annex to each language definition.

This Recommendation draws common elements from the meta-grammars of various languages, covering issues such as common lexical rules, the use of a universal character set, and syntax and constraint description for languages at both the abstract and concrete level.

Source

Recommendation ITU-T Z.111 was approved on 13 November 2008 by ITU-T Study Group 17 (2009-2012) under Recommendation ITU-T A.8 procedure.

FOREWORD

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

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

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

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

NOTE

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

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

INTELLECTUAL PROPERTY RIGHTS

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

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

CONTENTS

Introduction

Scope-objective: Notations and guidelines for the definition of formal languages are provided.

Coverage: The main features of this Recommendation are notations for defining the abstract and concrete syntaxes of languages, and a common structure for defining languages.

Applications: This Recommendation should be applied to new and possibly to revised formal language Recommendations.

Status/Stability: The notation and guidelines given are stable and have been used on some of the existing language Recommendations, such as Recommendations ITU-T Z.100 and Z.151. There is scope for further guidelines, extensions to the notations given, algorithms for conversions between the textual and metamodel presentations of abstract grammars, mappings to XML-based concrete syntaxes for interchange formats, and the possibility of further formal notations, such as a formal notation for specifying constraints.

Associated work: This Recommendation is generally associated with the study of formal languages for telecommunications applications in ITU-T, in particular the languages defined by the X.680 series, Z.100 series, Z.120 series, Z.130 series, Z.150 series and Z.160/Z.170 series of ITU-T Recommendations.

This Recommendation is also related to the unified modelling language (UML) and meta object facility (MOF) work of the object management group (OMG).

Background: Before the introduction of this Recommendation, the language Recommendations defined by ITU-T used different ways of describing the syntax and semantics of languages. This hinders the understanding, verifiability, and maintainability of these languages, and prevents their simple harmonization. Some language Recommendations have used grammars in Backus-Naur Form (BNF) to define concrete syntaxes and, in some cases, abstract syntaxes. More recent language definitions have used MOF metamodels to capture several aspects of languages. In both approaches, there has been a separation of abstract grammar from the concrete notation with a defined relationship between them. Each approach has its benefits and drawbacks. MOF metamodels are appealing due to their graphical nature (where associations and inherited concepts are explicit), whereas BNF grammars are easily analyzable by tools and there is considerable experience in their use.

This Recommendation gives guidelines for the definition of MOF-based metamodels describing languages in a way that is compatible with the approaches based on BNF grammars.

Traditional BNF-based grammars usually lack some of the capabilities of the MOF approach such as inheritance. MOF and UML-based meta-metamodels contain many features that make metamodels unnecessarily complex, difficult to understand semantically, and difficult to map to BNF grammars. This Recommendation focuses on a subset of modelling features that is expressive enough to describe language metamodels and that is isomorphic between the two approaches.

Recommendation ITU-T Z.111

Notations and guidelines for the definition of ITU-T languages

1 Scope

This Recommendation provides notation and guidelines for the definition of formal languages defined in new and optionally in revised language Recommendations that define formal languages or description techniques (such as those listed in 3.1.2 of [b-ITU-T Z.110], Criteria for use of formal description techniques by ITU-T).

1.1 Objective

The objective is to provide a basis for a common structure and meta-grammar notations to be used in formal language Recommendations, so that it is not necessary for each language Recommendation to describe its structure and the meta-grammars used: instead each language Recommendation can contain a reference to this Recommendation. By using a common structure and meta-grammar notations, it is also easier to integrate the different formal languages, therefore making it easier to use the languages together both with and without the support of tools. The use of a common structure and meta-grammar notations makes it easier to build tools that combine the ITU-T languages with each other and other notations. The use of a common structure and meta-grammar notations in different formal language Recommendations also makes it easier to understand several formal language Recommendations, because the structure and meta-grammar only has to be learnt once.

1.2 Application

When a new Recommendation is being drafted for a formal language, the application of this Recommendation should be applied. If it is decided not to apply this Recommendation, the reasons should be stated in the formal language Recommendation.

When a Recommendation for an existing formal language is being revised, the application of this Recommendation should be considered taking into account costs and benefits. If this Recommendation is not applied or is applied partially, the revised formal language Recommendation should at least contain a statement explaining that the structure and meta-language notations of the formal language Recommendation predated the approval of this Recommendation.

The structure and meta-grammar notations given in this Recommendation are also likely to be useful for the definition of other formal languages, not just formal languages defined in ITU-T Recommendations. If it is later decided to consider making a Recommendation for such a formal language, the prior application of this Recommendation will be a benefit.

2 References

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

[ITU-T T.55] Recommendation ITU-T T.55 (2008), *Use of the universal multiple-octet coded character set (UCS)*. <http://www.itu.int/rec/T-REC-T.55>

[OMG UML] *Unified Modeling Language: Superstructure, version 2.1.2, formal/2007-11-02 of the Object Management Group (OMG)* <http://www.omg.org/cgi-bin/doc?formal/07-011-02>.

3 Definitions

No specific definition is provided in this Recommendation.

4 Abbreviations and acronyms

This Recommendation uses the following abbreviations:

BNF Backus-Naur Form

MOF Meta Object Facility

OCL Object Constraint Language

OMG Object Management Group

SDL Specification and Description Language

UML Unified Modeling Language

XML eXtensible Markup Language

5 Conventions

If a language Recommendation conforms to this Recommendation, it conforms to the conventions defined in this clause and therefore these do not need to be repeated in the language Recommendation.

5.1 Grammars

A description only conforms with a Recommendation if it conforms to both the *Concrete grammar* and *Abstract grammar* of the corresponding language definition: that is, the description must be both recognizable as the language defined in the Recommendation and have the same meaning as defined by the *Semantics* in the Recommendation. If further concrete grammars are defined (in additional clauses, annexes or Recommendations), each of the concrete grammars has a definition of its own syntax and of its relationship to the abstract grammar (that is, how to transform into the abstract syntax). Using this approach, there is only one definition of the semantics of a language: the semantics of each of the concrete grammars is identified via its relationship to the abstract grammar. This approach also ensures that any further grammars are equivalent.

For some constructs of the concrete grammar, there may be no directly equivalent abstract syntax. In these cases, a *Model* is given for the transformation from concrete syntax into the concrete syntax of other constructs that (directly or indirectly via further models) have an abstract syntax. Items that have no mapping to the abstract syntax (such as comments) do not have any formal meaning.

5.2 Basic definitions

Some general concepts and conventions are used throughout a language Recommendation; their definitions are given in the following subclauses.

5.3 Presentation style

The following presentation style is used to separate the different language issues under each topic.

5.3.1 Division of text

A language Recommendation is organized by language features described by an optional introduction, which by convention shall be informative rather than normative (see clause 5.3.2), followed by titled enumeration items for:

a) *Abstract grammar* – Described by abstract syntax (either a textual grammar or a graphical metamodel) and static conditions (that is, static constraints) for a model in the language to be well-formed for an abstract model where artifacts of the concrete syntax are ignored.

b) *Concrete grammar* – Described by the graphical syntax, static conditions and rules for the graphical syntax to be well-formed (including drawing rules), the relationship of this syntax with the abstract syntax.

c) *Semantics* – Gives meaning to a construct, provides the properties it has, the way in which it is interpreted and any dynamic conditions that have to be fulfilled for the construct to behave well in the sense of the language defined.

d) *Model* – Gives the mapping for notations that do not have a direct abstract syntax and modelled in terms of other concrete syntax constructs. A notation that is modelled by other constructs is known as a shorthand, and is considered to be derived syntax for the transformed form.

To avoid misunderstandings, a language Recommendation that follows this Recommendation and uses informative introductions shall include a clause that states:

> Where a section has introductory text followed by a titled enumeration items (such as *Abstract grammar*, *Concrete grammar*, *Semantics*, *Model*) the introductory text before the first titled enumeration item in a clause is for information only, and is treated as if written as a NOTE (that is, informative not normative), except if it is explicitly stated as normative.

5.3.2 Titled enumeration items

Where a topic in the language definition has an introduction followed by a titled enumeration item, the introduction is considered to be an informal part of the Recommendation presented only to aid understanding and not to make that Recommendation complete.

If there is no text for a titled enumeration item (*Abstract grammar*, *Concrete grammar*, *Semantics*, *Model*), the whole item is omitted.

It is permitted for a Recommendation to define additional titled enumeration items, such as an *Example* section.

The remainder of this clause describes the other special formalisms used in each titled enumeration item and the titles used. It can also be considered as an example of the typographical layout of first-level titled enumeration items defined above where this text is part of an introductory section.

a) Abstract grammar

The notations for defining the abstract grammar are defined in clause 5.4.1. In the rest of this clause, the two notations are taken as equivalent. A rule in the textual abstract syntax (see clause 5.4.1.1) is equivalent to a model element in a graphical metamodel (see clause 5.4.1.2). The terms 'model element' and 'graphical metamodel' can be substituted for 'rule' and 'abstract syntax' respectively.

If the titled enumeration item *Abstract grammar* is omitted, then there is no additional abstract syntax for the topic being introduced, and the concrete syntax shall map onto the abstract syntax defined by another numbered text clause.

The rules in the abstract syntax may be referred to from any of the titled enumeration items by use of the rule name in italics.

The rules in the formal notation may be accompanied by paragraphs that define conditions which shall be satisfied by a well-formed piece (text and/or graphics) in the language being defined and which can be checked without dynamic interpretation of that piece. The static conditions at this point refer only to the abstract syntax. The static conditions shall be expressed in natural language, preferably supplemented by expression of the constraints in a more formal constraint language. Where it is possible to capture a constraint in the abstract syntax rather than by a static condition, this is preferable provided it does not make the syntax too complex. Static conditions, which are only relevant for the concrete syntax, are defined with the concrete syntax. Together with the abstract syntax, the static conditions for the abstract syntax define the abstract grammar of the language.

b) Concrete grammar

The concrete syntax shall be specified in the extended Backus-Naur Form of syntax description defined in clause 5.4.2, except where a graphical metamodel has been used for the abstract grammar, in which case it is permitted to extend the abstract metamodel to include concrete attributes. The use of natural language to define concrete syntax or definition by examples should be avoided.

The concrete syntax is accompanied by paragraphs defining the static conditions which must be satisfied in a well-formed definition and which can be checked without interpretation of a definition. Static conditions (if any) for the abstract grammar also apply. Where it is possible to capture a constraint in the concrete syntax rather than by a static condition, this is preferable provided it does not make the syntax too complex. The concrete grammar should not repeat constraints present in the abstract grammar.

In many cases there is a simple relationship between the concrete and abstract grammar, because the concrete syntax rule is simply represented by a single rule (or metamodel element) in the abstract grammar. When the same name is used in the abstract grammar and concrete syntax in order to signify that they represent the same concept, the text "<x> in the concrete syntax represents *X* in the abstract grammar" is implied in the language description and therefore does not need to be stated explicitly. In this context, spaces and hyphens are treated as equivalent, case is ignored but underlined semantic sub-categories (see clause 5.4.2) are significant, so that <<u>integer</u> name> represents *Integer-name* in the abstract syntax.

Concrete syntax that is not a shorthand form is strict concrete syntax. The relationship from concrete syntax to abstract syntax is defined only for the strict concrete syntax. Strict syntax is always defined in the *Concrete grammar*. The syntax for a shorthand is allowed to be defined in the *Model*.

The relationship between concrete syntax and abstract syntax is omitted if the topic being defined is a shorthand form that is modelled by other constructs of the defined language (see *Model* below).

When the name of a non-terminal ends in the concrete grammar with the word "diagram" and there is a name in the abstract grammar that differs only by ending in the word *definition*, then the two rules represent the same concept. For example, <system diagram> in the concrete grammar and *System-definition* in the abstract grammar correspond.

When the name of a non-terminal ends in the concrete grammar with the word "area" and there is a name in the abstract grammar that differs only by having the word *area* deleted, then the two rules represent the same concept. For example, <state partition area> in the concrete grammar and *State-partition* in the abstract grammar correspond.

c) *Semantics*

Properties are relations between different concepts in the defined language. Properties are used in the rules for a model to be well-formed.

An example (from the specification of SDL in [b-ITU-T Z.100]) of a property is the set of valid input signal identifiers of a process. This property is used in the static condition "For each *State-node*, all *Signal-identifier*s (in the valid input signal set) appear in either a *Save-signalset* or an *Input-node*".

Properties are static if they can be determined without interpretation of definitions in the defined language and are dynamic if an interpretation of the same is required to determine the property.

The interpretation is described in an operational manner. Whenever there is a list in the abstract syntax, the list is interpreted in the order given. That is, the Recommendation describes how objects of the semantic domain are created from a definition and how these objects are interpreted within an "abstract machine". Lists are denoted in the abstract syntax by the suffixes "*" and "+" (see clause 5.4.1).

Dynamic conditions are conditions to be satisfied during interpretation that cannot be checked without interpretation. Dynamic conditions may lead to errors.

NOTE – Behaviour of the object of the semantic domain is produced by "interpreting" the definition. The word "interpret" is explicitly chosen (rather than an alternative such as "executed") to include both mental interpretation by a human and the interpretation of the definition by a computer.

d) *Model*

Some constructs are considered to be "derived concrete syntax" (or a shorthand notation) for other equivalent concrete syntax constructs. For example, omitting an input for a signal is derived concrete syntax for an input for that signal followed by a null transition back to the same state.

The concrete syntax for a shorthand notation is allowed to be placed either in the *Concrete grammar* or in the *Model*.

The properties of a shorthand notation are derived from the way it is modelled in terms of (or transformed to) the primitive concepts. In order to ensure easy and unambiguous use of the shorthand notations, and to reduce side effects when several shorthand notations are combined, these concepts are transformed in a specified order.

The result of the transformation of a fragment of derived concrete syntax is usually either another fragment of derived concrete syntax, or a fragment of concrete syntax. The result of the transformation may also be empty. In the latter case, the original is removed from the specification.

Transformations can be inter-dependent and therefore the order in which various transformations are applied determines the validity and meaning of a definition.

5.4 Metalanguages

For the definition of properties and syntaxes of ITU-T languages, different metalanguages are used according to the particular needs.

In the following, an introduction of the metalanguages used is given.

5.4.1 Metalanguage for the abstract grammar

The abstract grammar of the language defines the relationships between elements of the language, without being concerned with issues such as punctuation marks that are needed to separate or terminate concrete syntax elements. There are two notations defined for the abstract grammar: a textual abstract syntax and a graphical metamodel notation.

One notation may not cover exactly everything found in the other notation, but the intention is that concrete grammar elements that map to the same abstract syntax elements have the same semantics.

5.4.1.1 Textual presentation

The following informally describes the textual presentation of the abstract syntax of ITU-T languages.

A definition rule in the abstract syntax can be regarded as a named composition defining a list of sub-components. The name starts with a letter (by convention uppercase) and is followed by any number of (by convention lowercase) letters and single hyphens ending in a (by convention lowercase) letter. When a name is used on the right-hand-side of a rule, it is allowed to be immediately followed by suffixes as described below.

To avoid confusion with the suffix "**-set**", a name shall not end in "-set". To avoid confusion with Quotation, a name should not be in all uppercase letters. The case of names is significant, so that *My-item* is distinct from *My-Item*, but the use of two names that differ only in the case of letters should be avoided, so that tools can be used to correct errors.

By convention names are in italics, which allows the rules and component names of the abstract syntax to be more easily recognized in the context of the other text and the name to be distinguished when used in plain text. The suffix "**-set**" by convention has the word "**set**" in bold.

A rule starts with the name of the rule followed by either "::" or "=". The rule continues on the same line and subsequent lines until followed by another rule or any printing character not allowed within a rule (such as a comma "," or a semi-colon ";"), or the end of the text containing the rules (for example, the end of a file, end of a text box, a heading, or a paragraph style not allowed for rules). Non-printing characters in rules are separators and are otherwise ignored, but by convention names of rules are always placed at the start of a new line, and continuation lines for a rule start with a non-printing character (usually a space or tab character). The characters that are allowed within a rule are ":" (part of "::"), "=", "*", "+", "[", "]", "|", "{", "}", "." (as part of ".." or "…"), "-", digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and (uppercase and lowercase) letters (A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z).

The symbol "::" can be read as "is defined as" and is used to define a domain in a definition rule.

For example:

Channel-path	::	*Originating-gate*
		Destination-gate
		*Signal-identifier**-set***

which defines the domain for the composition named *Channel-path.* This consists of three sub-components, which in turn are compositions or elementary domains (see below). It is allowed for a sub-component to contain the domain being defined, but (to avoid an infinitely recursive definition) there shall be at least one alternative of the domain definition that does not include such a sub-component.

It is allowed that a domain has no sub-components, and this is denoted by a "{}" as the right-hand-side of a definition rule. It is the presence (or absence) of an instance of such a domain that is still significant, and an instance of such a domain is distinct from an instance of any other domain even if the other domain does not have any components.

The symbol "=" can be read as "is equivalent to" and is used to define an equivalence rule where the name on the left-hand-side is equivalent to the syntactic expression on the right-hand-side. Wherever the equivalence name defined by the left-hand-side occurs, syntactically it can be replaced by the expression on the right-hand-side, grouped if the right-hand-side has multiple components. The reason for using "=" is to avoid repeating a common grouping especially in the case of alternatives, or to give a more specific name to a domain in a particular context to aid description of the language.

The definition

Agent-identifier	=	*Identifier*

expresses that an *Agent-identifier* is equivalent to an *Identifier* and therefore cannot (syntactically in the abstract syntax) be distinguished from other *Identifier* items defined in the same way such as a *Signal-identifier* where:

Signal-identifier	=	*Identifier*

The distinction between items defined as equivalent is not syntactic. For example, each *Identifier* instance has a value, which allows one instance to be distinguished from another and whether the value is the *Identifier* for a signal (*Signal-identifier*) or agent (*Agent-identifier*) in the model being defined. The name *Agent-identifier* is preferable wherever the *Identifier* has to identify an agent. The constraint (on the *Identifier* to identify an agent in this case) may be implied by general conventions or could be given by natural language and optionally by formal expression (in a language such as OCL). An alternative to naming equivalences is to name components of a domain (as described below).

The definition:

Agent-qualifier	::	*Agent-name*

expresses that an *Agent-qualifier* is a domain that has a component that is an *Agent-name* and *Agent-qualifier* is distinct from the domain for *Agent-name*. For example, if:

Agent-name = *Name*

Interface-name = *Name*

then an *Agent-qualifier* can be distinguished syntactically from an *Agent-name* and other items that are in the same domain as *Agent-name* such as an *Interface-name*.

The definition:

Nextstate-node = *Dash-nextstate* | *Named-nextstate*

expresses that a *Nextstate-node* is equivalent to the alternatives on the right-hand-side. Therefore syntactically,

Terminator :: *Nextstate-node*
| *Stop-node*
| …

has the same meaning as:

Terminator :: {*Dash-nextstate* | *Named-nextstate* }
| *Stop-node*
| …

An abstract syntax item might also be of some elementary (non-composite) domains. Predefined elementary domains are:

a) Natural

This is the elementary domain of non-negative integers.

Example:

Number-of-instances :: *Nat* [*Nat*]

Number-of-instances denotes a composite domain containing one mandatory natural (*Nat*) value and one optional natural ([*Nat*]) value denoting respectively the initial number and the optional maximum number of instances.

b) Boolean

Boolean denotes the domain of the two Boolean values **TRUE** and **FALSE**.

c) Quotation

These are represented as any bold face sequence of uppercase letters and digits. There can be a single quotation or a number of grouped quotations separated by "|", and in both cases this defines an enumerated domain with as many values as there are quotations. For example,

Agent-kind = **SYSTEM** | **BLOCK** | **PROCESS**

The quotation defines an enumerated domain with three values, represented by **SYSTEM**, **BLOCK** and **PROCESS**. Usually the names used in the abstract syntax correspond to the names in the concrete syntax, in this case the keywords **SYSTEM**, **BLOCK** and **PROCESS**. In this example, the name *Agent-kind* is made equivalent to the enumerated domain. If the quotation group was a component of a domain with other components (see example *Visibility-name* below), the enumeration domain has an implicit name only and cannot be reused in other contexts, and another use of the same quotations defines another implicit domain.

A single quotation corresponds to a domain with just one value, and will usually appear in an option. In such cases, it is usually preferable to replace the quotation with a Boolean.

Example:

Channel-definition :: *Channel-name*
[**NODELAY**]
*Channel-path-**set***

A channel definition has the property that it does not delay distinguished by an optional quotation **NODELAY**. This could be replaced by:

Channel-definition :: *Channel-name*
Is-delaying
*Channel-path-**set***
Is-delaying = *Boolean* := TRUE

Similarly if there are only two alternatives, this can usually be replaced by a Boolean. For example:

Category-definition	::	*Category-name* { **ABSTRACT** | **CONCRETE** } *Category-details*

could be replaced by:

Category-definition	::	*Category-name* *Is-concrete* *Category-details*
Is-concrete	=	*Boolean*

d) Token

Token denotes the domain of tokens. This domain can be considered to consist of a potentially infinite set of distinct atomic objects for which no representation is required.

Example:
Name :: *Token*

A name consists of an atomic object such that any *Name* can be distinguished from any other name.

e) Unspecified items

An unspecified item denotes domains which might have some representation, but for which the representation is of no concern in the language Recommendation.

Example:
Informal-text :: ...

Informal-text contains an object that does not have a formally defined content and structure and therefore is not formally interpreted. The meaning is therefore not formally defined, and if interpretation of the model leads to interpretation of the informal text, then the further behaviour of the system is not defined.

The following operators (constructors) of BNF (see clause 5.4.2) have the same use in the abstract syntax:

suffix "[n..m]" for a list of between n and m items, where n is a non-negative integer, m is a positive integer >= n;

suffix "[n]" for a list of exactly n items, where n is a positive integer – equivalent to the suffix [n..n];

suffix "[n..*]" for a list of at least n items, where n is a positive integer;

suffix "*" for a possibly empty list, which is equivalent to the suffix [0..*];

suffix "+" for a non-empty list, which is equivalent to the suffix [1..*];

separator "|" for an alternative;

brackets "[" "]" for meaning the enclosed item or items are optional.

Parentheses "{" and "}" are used for grouping of items that are logically related.

All the operators (including "-**set**" and "+**set**" optionally followed by the number of items – see below) are applied to either a domain name or to a grouping (within "{" and "}").

A domain is allowed to be specified to have a structure derived from another domain, in which case we say the former (child) domain inherits its structure from the latter (or parent) domain. A child domain has all the sub-components defined by the parent domain, and it is allowed to add sub-components which follow the inherited components. A child item object is allowed in any place of the tree of objects where parent item objects are allowed.

For example, the definition:

Agent-identifier (Agent-name) :: *Qualifier*

defines an *Agent-identifier* to have the same structure as *Agent-name* but has another component that is a *Qualifier*. As far as the structure is concerned, the above definition is the same as defining:

Agent-identifier :: *Agent-name Qualifier*

However, the former inheritance definition means that an *Agent-identifier* instance is allowed wherever an *Agent-name* instance is allowed, so rather than specify the alternatives { *Agent-identifier* | *Agent-name* }, a single *Agent-name* is sufficient.

Finally, the abstract syntax uses two other postfix operators "-**set**" yielding a possibly empty set (unordered collection of distinct objects) and the postfix operator "+**set**" yields a set which shall not be empty. These can also have the suffixes "[n]" for a set of exactly n items, suffix "[n..m]" for a set of between n and m items, suffix "[n..*]" for a set of at least n items.

Example:
Agent-graph :: *Agent-start-node State-node-**set***

An *Agent-graph* consists of an *Agent-start-node* and a possibly empty set of *State-nodes*.

Where a sub-component is of an elementary domain, or a quotation, a default value can be specified for this sub-component. For example,

Number-of-instances :: *Nat*:=1 [*Nat*]

specifies that the first *Nat* component will have the value 1, if no value is specifically given in the concrete notation.

Visibility-name :: *Name* { **PUBLIC** | **PROTECTED** | **PRIVATE** := **PUBLIC** }

specifies that the second component will have the Quotation value **PUBLIC**, if no value is specifically given in the concrete notation.

Suffixes that specify multiplicity for a list or set specify constraints on the grammar if the lower bound is greater than zero or if an upper bound is given (that is, it is not "*"). Often there are other constraints that need to be specified: for example, in *Number-of-instances* if there is a second *Nat*, this represents the maximum number of instances and has to be greater than or equal to the initial number of instances given by the first *Nat*. A natural language description of such constraints helps explain the reason for the limitation, but writing natural language so that the formulation of the constraint is clear and unambiguous can be difficult, especially if the text is to be translated into different natural languages. For that reason, it is desirable that constraints on the grammar are written in a formal notation (such as OCL). To be able to clearly refer to the domain components of a domain, each domain component can optionally be given a name starting with a lowercase letter preceding the component domain by a colon, for example:

Number-of-instances :: *initialNumber*: *Nat*:=1 [*maximumNumber*: *Nat*]

In this case, it is now possible to write *Number-of-instances.maximumNumber >= Number-of-instances.initialNumber*. An alternative to named domain components is to introduce an equivalence rules for each component. For example:

Number-of-instances :: *Initial-number* [*Maximum-number*]

Initial-number = *Nat* :=1

Maximum-number = *Nat*

The main differences are the relative conciseness of the named component notation, and that a component name is local to a domain whereas an equivalence name can be used in several domains.

The following gives a more formal abstract description of the textual abstract grammar defined in the notation itself as far as possible and supplementary to the text above:

Grammar :: *Rule*-**set**

Rule :: *Definition-rule* | *Equivalence-rule*

Definition-rule :: *Name* [*Non-terminal-domain*] [*Expression*]

Equivalence-rule :: *Name Expression*

Expression :: *Alternatives* | *Composition* | *Option* | *List* | *Set* | [*Component-name*] *Domain*

Alternatives :: *Expression*[2..*]

Composition :: *Expression*[1..*]

Option :: *Expression*

List :: *Expression minimum*: *Nat*:=0 [*maximum*: *Nat*]

Set :: *Expression minimum*: *Nat*:=0 [*maximum*: *Nat*]

Domain :: *Non-terminal-domain*
| *Elementary-domain*
| *Enumeration-domain*

Non-terminal-domain :: *Name*

Elementary-domain	:: *Elementary-domain-kind* [*Default-elementary-value*]
Elementary-domain-kind	= **NAT | TOKEN | BOOLEAN | UNSPECIFIED**
Default-elementary-value	= *Token*
Enumeration-domain	:: *Quotation*-**set** [*Default-enumeration-value*]
Default-enumeration-value	:: *Quotation*
Component-name	= *Token*
Name	= *Token*

There should normally be one *Rule* from which every other *Rule* can be reached.

The same character string always maps to the same *Token* value, and two different character strings map to different *Token* values. The character strings *Token*, *Nat*, *Boolean* and *Informal-text* shall not be used for a *Name*, because these are reserved for each of the respective domains.

A *Name* is a *Token* that is represented by a character string starting with a letter, followed by any number of alphanumeric characters and single hyphens and terminating in an alphanumeric. Each *Name* of a *Definition-rule* or *Equivalence-rule* shall have a *Token* value distinct from the *Token* value of the *Name* of any other *Definition-rule* or *Equivalence-rule*.

A *Non-terminal-domain* has a *Token* that is represented by name starting with a letter, followed by any number of alphanumeric characters and single hyphens and terminating in an alphanumeric. The *Name* of a *Non-terminal-domain* shall have the same *Token* value as the *Name* of a *Definition-rule* or *Equivalence-rule*.

NAT, **TOKEN**, **BOOLEAN** and **UNSPECIFIED** are represented by *Token, Nat*, *Boolean* and *Informal-text* respectively.

A *Default-elementary-value* is a *Token* that is represented in the case of *Nat* (**NAT**) by a non-negative integer (a string of digits) and in the case of *Boolean* (**BOOLEAN**) by one of the strings TRUE or FALSE. If the *Elementary-domain-kind* of an *Elementary-domain* is *Token* or *Informal-text*, then assignment is not allowed and the *Default-enumeration-value* shall be omitted. If the *Elementary-domain-kind* of an *Elementary-domain* is *Nat*, then *Default-enumeration-value* shall be a *Token* for an integer. If the *Elementary-domain-kind* of an *Elementary-domain* is *Boolean*, then *Default-enumeration-value* shall be a *Token* for a Boolean value.

Quotation is represented by bold face sequence of uppercase letters and digits, called a quotation in this paragraph. The same quotation always maps to the same *Quotation* value and two different quotations map to different *Quotation* values. The quotation for a *Quotation* shall not be the same as the character string for a *Token*. An *Enumeration-domain* is represented either by a single quotation or by a list of quotations separated by "|" characters. When a "|" character is between two quotations, it is a separator for an *Enumeration-domain* rather than for *Alternatives*. The *Default-enumeration-value* (if present) of an *Enumeration-domain* shall be one of the *Quotation* values in the *Quotation*-**set** of the *Enumeration-domain.*

Alternatives or *Composition* within an *Expression* that is *Alternatives* or *Composition* or *List* or *Set* are grouped by "{" and "}" brackets in the concrete notation.

The *minimum Nat* for a *List* or *Set* is the minimum number of elements, and the *maximum Nat* is the maximum number. If a *maximum* exists, it must be greater than or equal to the *minimum*. If *maximum* is omitted, there is no maximum number of elements. The notation for a list or set is explained above.

5.4.1.2 Metamodel presentation

The following describes an alternative to the textual presentation introduced in clause 5.4.1.1. This metamodel representation of the abstract syntax of ITU-T languages is explained using a meta-metamodel (Figure 1).

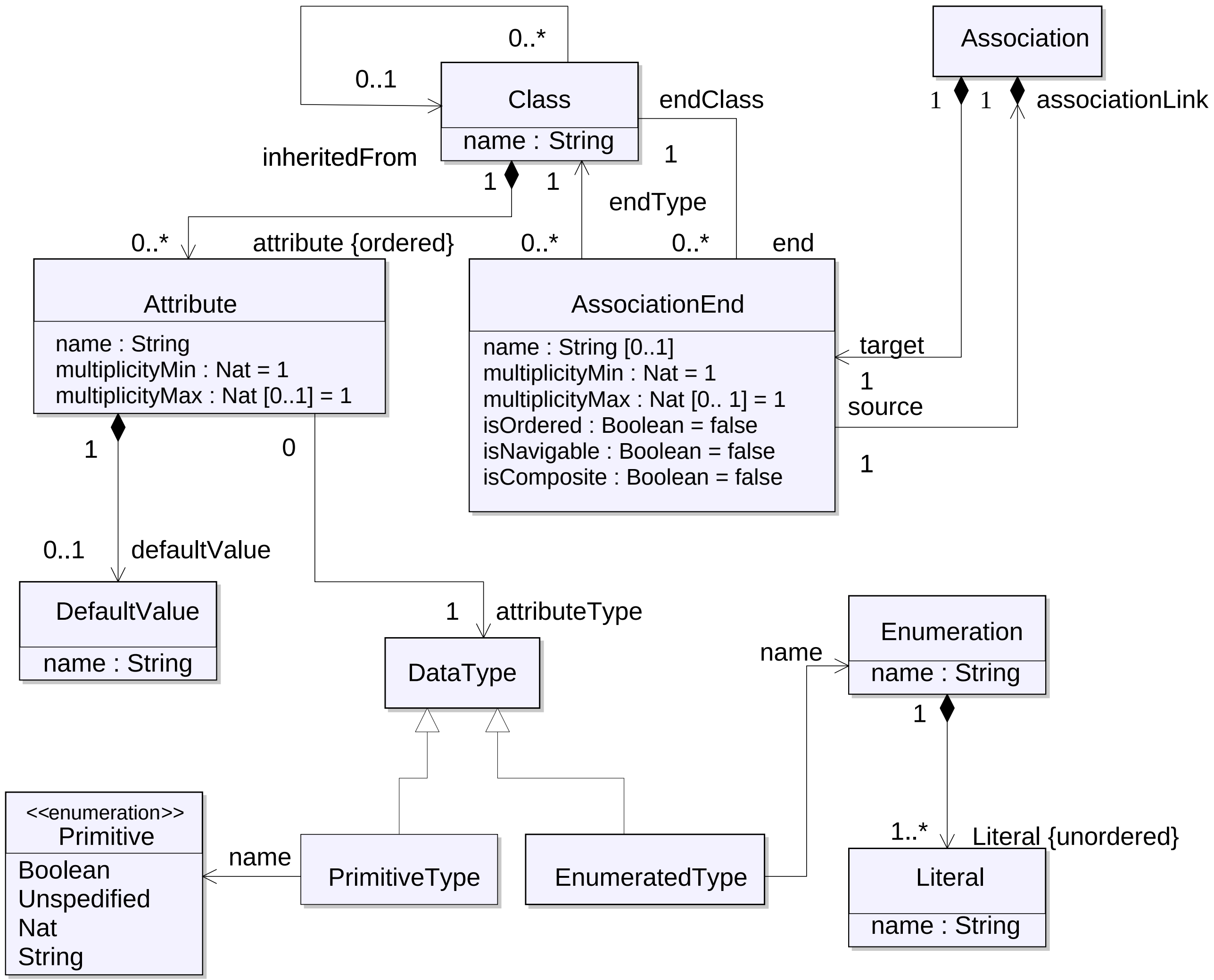


Figure 1 – Meta-metamodel for metamodeling language elements

The meta-metamodel elements described below are represented using UML in Figure 1, except where otherwise noted. Note that, in order to better align metamodels based on Z.111 with MOF and UML, *Token* elements are shown with the "String" keyword in metamodels, including the meta-metamodel in Figure 1.

In the following, a Named element is a meta-class that contains a *name* attribute of the *Token* primitive type (shown visually as String). Some *Token* values represent Nat values (that is non-negative Integers), and two distinct *Token* values represent the Boolean true and false values. These *Token* values for Nat and Boolean shall only be used in the *value* of a *DefaultValue* and shall not be used for the *Token* value of any *name*. When deriving the *Token* from a character string in the concrete notation, the *Token* value depends on the character string and the case of letters in the character string are taken into account. The same character string in the concrete notation for names shall always produce the same *Token*, so that constraints on *Token* values are constraints on character string in the concrete notation for the *name* attributes. Two *name* attributes are the same if they have the same *Token* value, otherwise they are distinct.

i) *Class* is a Named element that contains one or more *Attribute* items and can participate in binary *Association* relations via one or both of the *AssociationEnd* attributes of each *Association*. A *Class* can inherit the *Attribute* and *AssociationEnd* (and hence *Association*) items from another *Class* (single inheritance only).

   Each *Class* shall have a *name* with a *Token* value that is distinct from the *Token* value for the *name* of any other *Class* (including predefined primitives), or any *Enumeration*.

The *inheritedFrom* attribute (if present) of *Class* references the parent *Class* of a *Class*. An inherited *Class* of a *Class* is either the parent *Class* or any inherited *Class* of the parent *Class*, transitively. A *Class* shall not have itself as an inherited *Class*. The *attribute list* of a *Class* includes the *attribute* list of the parent *Class*, so that the same *Attribute* shall be referenced by each element of the *attribute* list of the parent *Class* and the corresponding (by order) element of the *attribute* list of the *Class*. A *Class* that inherits from a parent *Class* includes each *end* reference to an *AssociationEnd* of parent *Class* and is the *endClass* of the *AssociationEnd*. A *Class* that inherits from a parent *Class* is referenced as an *endType* by each *AssociationEnd* that references the parent *Class* as an *endType*. Multiple classification is not allowed, that is, an instance of a *Class* C1 cannot be an instance of another *Class* C2, if C1 is not inherited (directly or transitively) from C2.

The *name* (if present) of each *AssociationEnd* that has an *end* in a given *Class* shall be distinct from the *Token* value for the *name* (if present) of any *AssociationEnd* that has an *end* in the *Class* or any inherited *Class*.

ii) *Attribute* is a Named element. It is a meta-class for an ordered element of a *Class* that has a *DataType* and an optional *DefaultValue*.

The *name* of each *attribute* of a given *Class* shall be distinct from the *name* of any other *attribute* of the *Class* or any inherited *Class* and shall also be distinct from the *name* (if present) of any *AssociationEnd* that has an *end* in the *Class* or any inherited *Class*.

The multiplicity of an *Attribute* is given by *multiplicityMin* and *multiplicityMax*, which have default values of 1 if they are omitted in concrete notation. However, if in the concrete notation the maximum is given by "*", the maximum is empty and there is no upperbound on the number of *Attribute* instances. If *multiplicityMax* is present, it shall be greater than 0 and also greater than or equal to *multiplicityMin*.

If the *Attribute* has a *DataType* that is a *PrimitiveType* Token or *PrimitiveType* Unspecified, the *DefaultValue* shall be absent. If a *DefaultValue* is present, its *Token* value shall represent a value of the *DataType* of the *Attribute* (that is, it shall be true or false for a Boolean, a non-negative integer for a Nat, or a quotation *Literal* for an *Enumeration*).

iii) *DataType* is a meta-class that is either a *PrimitiveType* or an *EnumeratedType*.

iv) *PrimitiveType* is a *DataType* where the type is predefined (one of *Token*, *Nat*, *Boolean*, *Unspecified*) primitive identified by the *Primitive* value of the *name* attribute of the *PrimitiveType*. Unspecified is a primitive that might have some representation, but for which the representation is of no concern in the language Recommendation.

v) *EnumeratedType* is a *DataType* where the type is an *Enumeration* identified by its *name* attribute.

vi) *Enumeration* is a Named element that represents a (quotation) *DataType* whose values are represented by a set of *literal* elements.

Each *Enumeration* shall have a *name* with a *Token* value that is distinct from the *Token* value for the *name* of any other *Enumeration* or any *Class* (including predefined primitives).

vii) *Literal* is a Named element contained in an *Enumeration* that is one of the values of an *Enumeration*. The *name* of each *literal* of an *Enumeration* shall be distinct for any other *literal* of the *Enumeration*, but it is permitted for the *name* to be the same as the *literal* of another *Enumeration*.

viii) *DefaultValue* is a meta-class for the optional component of an *Attribute* that identifies a default value of the appropriate type (see *Attribute*).

ix) *Association* is a meta-class for the relation between *Class* meta-classes. It has two *AssociationEnd* attributes (*source* and *target*) for the logical connection of the *Association* with the related *Class* meta-classes.

An *Association* shall have at least one *AssociationEnd* that is navigable (*isNavigable* = true). An *Association* is represented by a line in the concrete notation. An *Association* navigable in only one direction is represented by a line with an arrow on the navigable end.

x) *AssociationEnd* is a meta-class for the end of an *Association*. It has Boolean attributes that determine if the *AssociationEnd* is composite (*isComposite*), navigable (*isNavigable*) or ordered (*isOrdered*). If the *AssociationEnd* is navigable (*isNavigable* = true), the *name* shall not be empty, and the *name* (which corresponds to the role name of the *AssociationEnd*) shall be distinct from the *name* of any other *AssociationEnd* that is an *end* of the *Class* that is the *source* of the *AssociationEnd*.

An *AssociationEnd* shall not be both the *source* and *target* of an *Association*. In the following, an *AssociationEnd* that is the *source* of an *Association* is called a source *AssociationEnd*, and an *AssociationEnd* that is the *target* of an *Association* is called a target *AssociationEnd*.

The *Class* of the *endType* of a source *AssociationEnd* is derived and is the same as the *Class* of the *endClass* of the target *AssociationEnd* of the *Association* that is the *source* of the source *AssociationEnd*. The *Class* of the *endType* of a target *AssociationEnd* is also derived and is the same as the *Class* of the *endClass* of the source *AssociationEnd* of the *Association* that is the *target* of the target *AssociationEnd*. More formally:

source *AssociationEnd.endType* = *AssociationEnd.associationlink.target.endClass*; and

target *AssociationEnd.endType* = *AssociationEnd.associationlink.source.endClass*.

A source *AssociationEnd* is allowed to be composite (*isComposite* = true), and in this case, the *Class* with the source *AssociationEnd* as an *end* has as a component an item (or a collection of items – see multiplicity) with the *endType* of the source *AssociationEnd*. A *Class* with a non-composite source *AssociationEnd* is related (by the *Association*) to an item or collection with the *endType* of the source *AssociationEnd*. In the concrete notation, an *AssociationEnd* that is composite is represented by filled diamond at the *AssociationEnd* that is the *source*.

The *AssociationEnd* that is the *target* of an *Association* shall not be composite (*isComposite* = false).

An *AssociationEnd* has a multiplicity defined by the values of the *multiplicityMin* and *multiplicityMax*, which are natural numbers. Although default values are given for the values, it is not allowed to omit the values in the concrete notation. If, in the concrete notation, the maximum is given by "*", the maximum is empty and there is no upperbound on the number of instances. If *multiplicityMax* is present, it shall be greater than 0 and also greater than or equal to *multiplicityMin*. If *multiplicityMax* is equal to *multiplicityMin*, then it is sufficient in the metamodel representation to show only one number (e.g., 1 instead of 1..1).

If *multiplicityMax* is larger than 1 or omitted, the *AssociationEnd* denotes a collection of items. By default, this collection is ordered (*isOrdered* = true), so the collection is a list. If the collection is not ordered (*isOrdered* = false), the collection will normally be a set (but is allowed to be a bag). In a set, each item in the collection has a different value. In a bag, there may be any number of items with the same value.

Several modelling elements commonly used in class diagrams should be avoided when describing metamodels for abstract syntaxes. These include:

– *Packages*: A structuring mechanism that does not add semantic value at the abstract level.

– *Visibility of attributes and association ends*: This has no impact on the abstract model as it is hidden.

– *Absence of association multiplicities*: To avoid ambiguities in the understanding of default multiplicities (according to standards or people), multiplicities at association ends are required to be explicit.

– *Multiple inheritance*: May require name resolution and hence should be avoided.

– *Operations*: Not usually needed in the abstract model. Operations on model elements (in a language such as OCL) are useful to describe constraints formally, but constraints should also be expressed in natural language as well as formally so the intent can be captured.

– *Abstract classes*: Not needed because the mapping between a concrete syntax and an abstract syntax can prevent the instantiation of such classes.

– *Interfaces*: Not needed since there are no operations on instances.

– Do not include layout elements (graphical information such as colour, positions, shapes, sizes).

5.4.2 Metalanguage for the concrete grammar

The syntax of the concrete grammar is defined by the extended Backus-Naur Form (BNF) defined in more detail in this clause. If necessary, syntax is supplemented by further definition in natural language and preferably also in a formal language such as OCL. The concrete syntax is mapped to the abstract syntax (either implicitly as described in clause 5.3.2 b) *Concrete Grammar* above or explicitly) and the constraints and semantics of the abstract grammar apply.

The grammar should usually be separated into two levels:

1) lexical rules that define the syntax of lexical units such as names, numbers, character strings, composite symbols (such as "=>") and any reserved keyword names of the language;

2) non-lexical rules that use the lexical rules as terminal symbols.

If the grammar is simple, it is acceptable to include the lexical rules with the non-lexical rules. The definition of graphical symbols (such as a frame box or a flow line) are treated like keywords and defined in the lexical rules, or as part of the non-lexical grammar.

In the BNF for lexical rules, the terminals are <space> and the printed characters are specified as terminal symbols by the defined language. The printed characters are not limited to the Latin alphabet; the universal multiple-octet coded character set defined in [ITU-T T.55] is allowed to be used. The actual printed character set used and non-printing characters such as <space> are defined in the formal language Recommendation. Some examples of lexical rules are:

```
<asterisk>          ::=   *

<bit string>        ::=
                          <apostrophe> { 0 | 1 }* <apostrophe> { B | b }
```

NOTE – It is assumed that characters are ordered and presented left to right then top to bottom as in Latin scripts. If other orderings (such as right to left, or top to bottom, mixed ordering) are significant, the relationship between the order characters is analysed and how they are presented has to be defined.

In the Backus-Naur Form for non-lexical rules, a terminal symbol is one of the lexical units defined to be terminal. In non-lexical rules, a terminal can be represented by one of the following:

a) a keyword (such as **state**);

b) the character for the lexical unit if it consists of a single character (such as "=");

c) the lexical unit name (such as <quoted operation name> or <bit string>);

d) the name of a <composite special> lexical unit (such as <implies sign> which is defined as "=>").

To avoid confusion with BNF grammar, the lexical unit names <asterisk>, <plus sign>, <vertical line>, <left square bracket>, <right square bracket>, <left curly bracket> and <right curly bracket> are always used rather than the equivalent characters (except in the lexical rule that defines the unit – see the example of <asterisk> above). Note that non-lexical terminals that have multiple instances (typically the lexical units for <name> and <character string>) are permitted to have semantics stressed as defined below.

The angle brackets and enclosed word(s) are either a non-terminal symbol or one of the lexical units. Syntactic categories are the non-terminals indicated by one or more words enclosed between angle brackets. For each non-terminal symbol, a production rule is given in the concrete grammar. For example,

<block reference> ::=

block <block name> **referenced** <end>

A production rule for a non-terminal symbol consists of the non-terminal symbol at the left-hand side of the symbol "::=", and one or more constructs, consisting of non-terminal and/or terminal symbol(s) at the right-hand side. For example, <block reference>, <block name> and <end> in the example above are non-terminals; **block** and **referenced** are terminal symbols. By convention, a rule always starts on a new line and continuation lines start with a non-printing character (usually a space or tab). The rule continues on the same line and subsequent lines until followed by the start of another rule (a line starting with non-terminal symbol followed by "::=") or the end of the text containing rules (indicated by text style, a heading, characters not allowed in rules or some other means).

Sometimes the symbol includes an underlined part. This underlined part stresses a semantic aspect of that symbol. For example, <block name> is syntactically identical to <name>, but semantically it requires the name to be a block name.

At the right-hand side of the "::=" symbol, several alternative productions for the non-terminal can be given, separated by vertical bars ("|"). For example,

```
<diagram in package> ::=
            <package diagram>
        |   <package reference area>
        |   <entity in agent diagram>
        |   <data type reference area>
        |   <signal reference area>
        |   <procedure reference area>
        |   <interface reference area>
        |   <create line area>
        |   <option area>
```

expresses that a <diagram in package> is a <package diagram>, or a <package reference area>, or an <entity in agent diagram>, or a <data type reference area>, or a <signal reference area>, or a <procedure reference area>, or an <interface reference area>, or a <create line area> or an <option area>.

Syntactic elements may be grouped together by using curly brackets ("{" and "}"), similar to the parentheses in the textual presentation of abstract syntax (see clause 5.4.1.1). A curly bracketed group may contain one or more vertical bars, indicating alternative syntactic elements.

Repetition of syntactic elements or curly bracketed groups is indicated by:

suffix "[n..m]" the group is repeated between n and m items, where n is an integer >=0, m is a positive integer >= n;

suffix "[n]" the group is repeated exactly n items, where n is a positive integer – equivalent to the suffix [n..n];

suffix "[n..*]" the group is repeated at least n items, where n is a positive integer;

suffix "*" the group is optional and repetition of any number of times is allowed – equivalent to the suffix [0..*];

suffix "+" the group shall be present and repetition of any number of times is allowed – equivalent to the suffix [1..*].

For example,

<operation definitions> ::=

{ <operation definition>
| <operation reference>
| <external operation definition> }+

The example above expresses that <operation definitions> may contain zero or more definitions of <operation definition> or <operation reference> or <external operation definition>, and may contain more than one of any of these.

A repetition suffix can include a list separator after the repetition. If the repetition uses square brackets ("[" and "]"), a vertical bar ("|") is placed before the closing square bracket ("]") and the separator is placed between the vertical bar ("|") and the closing square bracket ("]"). If the repetition suffix is an asterisk ("*") or plus sign ("+"), it is followed by an opening square bracket and vertical line pair ("[|") the separator and then a closing square bracket ("]"). For example,

<two or more declarations> ::=

{ <name>+[|,] : <type> }[2,*| ;]

The example above expresses that <declaration list> contains at least two and possibly many repetitions separated by semicolons of the sequence: <name> list separated by commas, colon and <type>. This is equivalent to the following syntax that uses plain repetition (without separators) where <name> is repeated:

<two or more declarations> ::=

<name> { , <name> }* : <type>

{ ; <name> { , <name> }* : <type> }+

This can also be expressed using recursion instead of repetition as:

<two or more declarations> ::=

<name list> : <type> ; { <name list> : <type> | < two or more declarations> }

<name list> ::=

{ <name> | <name> , <name list> }

If syntactic elements are grouped using square brackets ("[" and "]"), then the group is optional. For example,

<valid input signal set> ::=

signalset [<signal list>] <end>

expresses that a <valid input signal set> may, but need not, contain <signal list>. An optional grouping is equivalent to a repetition suffix [0..1].

If there is any ambiguity between an optional group and the use of square brackets ("[" and "]") for a repetition suffix, it is a repetition unless there is at least one layout character (such as a space) before the "[" in which case it is an optional group.

To support the graphical grammar, the metalanguage has the following metasymbols:

a) ***set***

b) ***contains***

c) ***is associated with***

d) ***is followed by***

e) ***is connected to***

f) ***is attached to***

The ***set*** metasymbol is a postfix operator operating on the immediately preceding syntactic elements within curly brackets, and indicating an (unordered) set of items. Each item may be any group of syntactic elements, in which case it must be expanded before applying the ***set*** metasymbol.

Example:

{ <operation text area>* <operation body area> } ***set***

is a set consisting of zero or more <operation text area>s, and one <operation body area>. The ***set*** metasymbol is used when the position of the syntactic elements relative to one another in the diagram is irrelevant and the elements can be considered in any order.

All the other metasymbols are infix operators, having a graphical non-terminal symbol as the left-hand argument. The right-hand argument is either a group of syntactic elements within curly brackets or a single syntactic element. If the right-hand side of a production rule has a graphical non-terminal symbol as the first element and contains one or more of these infix operators, then the graphical non-terminal symbol is the left-hand argument of each of these infix operators. A graphical non-terminal symbol is a non-terminal ending with the word "symbol".

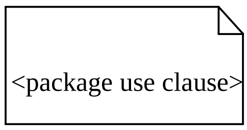
The metasymbol ***contains*** indicates that its right-hand argument should be placed within its left-hand argument and the attached <text extension symbol>, if any. The right-hand argument is expanded within the symbol, should not cross the symbol boundaries and is distinct from any occurrence of the same syntax in another rule. For example,

<package use area> ::=

<text symbol> contains <package use clause>

<text symbol> ::=

means the following



The metasymbol ***is associated with*** indicates that its right-hand argument is logically associated with its left-hand argument (as if it were "contained" in that argument, the unambiguous association is ensured by appropriate drawing rules). The right-hand argument is expanded and is distinct from any occurrence of the same syntax in another rule.

The metasymbol ***is followed by*** means that its right-hand argument follows (both logically and in drawing) its left-hand argument. The right-hand argument is expanded at the end of the implied symbol, and is distinct from any occurrence of the same syntax in another rule.

The metasymbol ***is connected to*** means that its right-hand argument is connected (both logically and in drawing) to its left-hand argument. The right-hand argument is expanded, and is distinct from any occurrence of the same syntax in another rule (in contrast to ***is attached to*** below).

The metasymbol ***is attached to*** expresses syntax requirements but not syntax productions. The metasymbol ***is attached to*** requires its right-hand argument and left-hand argument be attached to each other (both logically and in drawing), but one argument is not expanded with the syntax for the other argument, but each shall exist as separate expansions from syntax rules (in contrast to ***is connected to*** above). Being attached is mutual, so that A ***is attached to*** B is always matched in the syntax by another rule where B ***is attached to*** A, though this need not be directly expressed on B. For example, B may have alternatives B1 and B2 each of which ***is attached to*** A. Being attached will usually mean that the abstract syntax for each side contains the identifier of the other side.

Appendix I

Mapping from the metamodel presentation to the textual presentation: High-level description

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

The metamodel presentation and the BNF-based textual presentation of the abstract syntax each have benefits and drawbacks. Metamodels are appealing due to their graphical nature (where associations and inherited concepts are explicit), whereas BNF grammars are easily analysable by tools and there is considerable experience in their use. A procedure is outlined here that converts an abstract syntax expressed as a metamodel into an abstract syntax expressed textually. Leveraging this procedure, an abstract syntax can be expressed as a metamodel for convenience, but can be subject to validation through grammar-oriented tools.

The following gives the mapping from the concepts in the metamodel presentation of the abstract syntax to the textual representation of the abstract syntax.

A *Class* corresponds to a *Definition-rule*. The *name* corresponds to the *Name*. The class referenced by *inheritsFrom* corresponds to the *Non-terminal-domain*, if any. The set of *attribute* and association *end* corresponds to the *Expression*.

If the *Attribute* or *AssociationEnd* is named, this corresponds to an *Equivalence-rule*, where *Name* is derived from *name* and the current class *name* and the attribute or association end constructs the *Expression* of the alias as in the following, and *name* corresponds also to *Symbol* in the outer *Expression*. If the *name* is unique, it corresponds to *Token*. Otherwise, the *Expression* is constructed as follows.

If *multiplicityMin* of an *Attribute* or *AssociationEnd* is 0 and *multiplicityMax* of this *Attribute* or *AssociationEnd* is 1, and *isNavigable* is true, this corresponds to an *Option*, where *Expression* is a *Symbol* corresponding to the *type* of the attribute or association end.

If *multiplicityMin* of an *Attribute* or *AssociationEnd* is 0, and *multiplicityMax* of this *Attribute* or *AssociationEnd* is *, and *isOrdered* of this *Attribute* or *AssociationEnd* is true, and *isNavigable* is true, this corresponds to a *List*, where *Expression* is a *Symbol* corresponding to the *type* of the attribute or association end.

If *multiplicityMin* of an *Attribute* or *AssociationEnd* is 1, and *multiplicityMax* of this *Attribute* or *AssociationEnd* is *, and *isOrdered* of this *Attribute* or *AssociationEnd* is true, and *isNavigable* is true, this corresponds to a *List* with a minimum of one element, where *Expression* is a *Symbol* corresponding to the *type* of the attribute or association end.

If *multiplicityMin* of an *Attribute* or *AssociationEnd* is 0, and *multiplicityMax* of this *Attribute* or *AssociationEnd* is *, and *isOrdered* of this *Attribute* or *AssociationEnd* is false, and *isNavigable* is true, this corresponds to a *Set*, where *Expression* is a *Symbol* corresponding to the *type* of the attribute or association end.

If *multiplicityMin* of an *Attribute* or *AssociationEnd* is 1, and *multiplicityMax* of this *Attribute* or *AssociationEnd* is *, and *isOrdered* of this *Attribute* or *AssociationEnd* is false, and *isNavigable* is true, this corresponds to a *Set* with a minimum of one element where *Expression* is a *Symbol* corresponding to the *type* of the attribute or association end.

If *multiplicityMin* of an *Attribute* or *AssociationEnd* is 1 and *multiplicityMax* of this *Attribute* or *AssociationEnd* is 1, and *isNavigable* is true, this corresponds to a *Symbol*, such that the *Symbol* is corresponding to the *type* of the attribute or association end.

If *isComposite* of an *AssociationEnd* is false, then the *Symbol* is a *Name*; otherwise, the *Symbol* is a *Non-terminal-domain*.

If an *Attribute* has a *default*, then this generates an *Assignment*, where *DefaultValue* corresponds to *Elementary-domain* and the *name* of the type of the *Attribute* corresponds to *Token*.

Any *multiplicityMin* and *multiplicityMax* on an *AssociationEnd* where *isNavigable* is false and which is not both 1 in the case where *isComposite* is true or is otherwise 0 and *, respectively, generates a constraint on the respective element.

An *Enumeration* corresponds to a *Non-terminal-domain* where the *name* corresponds to *Token*, and *Expression* is an *Alternative* where each *Symbol* is an *Elementary-domain* corresponding to the *literals*.

Bibliography

[b-ITU-T Z.100] Recommendation ITU-T Z.100 (2007), *Specification and Description Language (SDL)*. <http://www.itu.int/rec/T-REC-Z.100>

[b-ITU-T Z.110] Recommendation ITU-T Z.110 (2008), *Criteria for use of formal description techniques by ITU-T*. <http://www.itu.int/rec/T-REC-Z.110>

[b-OMG-OCL] *Object Constraint Language Specification, Version 2.0, formal/2006-05-01 of the Object Management Group (OMG)*. <http://www.omg.org/cgi-bin/doc?formal/2006-05-01>

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects and next-generation networks
Series Z	**Languages and general software aspects for telecommunication systems**

Printed in Switzerland
Geneva, 2009