INTERNATIONAL STANDARD

ISO 24610-2

First edition
2011-10-01

Language resource management — Feature structures —

Part 2: Feature system declaration

Gestion des ressources langagières — Structures de traits —

Partie 2: Déclaration de système de structures de traits

Reference number
ISO 24610-2:2011(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO 24610-2 was prepared by Technical Committee ISO/TC 37, *Terminology and other language and content resources*, Subcommittee SC 4, *Language resource management*.

ISO 24610 consists of the following parts, under the general title *Language resource management — Feature structures*:

— *Part 1: Feature structure representation*

— *Part 2: Feature system declaration*

Introduction

ISO 24610 is organized in two separate main parts.

— Part 1, *Feature structure representation*, is dedicated to the description of feature structures, providing an informal and yet explicit outline of their characteristics, as well as an XML-based structured way of representing feature structures in general and typed feature structures in particular. It is designed to lay a basis for constructing an XML-based reference format for exchanging (typed) feature structures between applications.

— Part 2, *Feature system declaration*, will provide an implementation standard for XML-based typed feature structures, first by defining a set of types and their hierarchy, then by formulating type constraints on a set of features and their respective admissible feature values and finally by introducing a set of validity conditions on feature structures for particular applications, especially related to the goal of language resource management.

A feature structure is a general-purpose data structure that identifies and groups together individual features by assigning a particular value to each. Because of the generality of feature structures, they can be used to represent many different kinds of information. Interrelations among various pieces of information and their instantiation in markup provide a meta-language for representing linguistic content. Moreover, this instantiation allows a specification of a set of features and values associated with specific types and their restrictions, by means of feature system declarations, or other XML mechanisms to be discussed in this part of ISO 24610.

Some of the statements here are copied from ISO 24610-1:2006 in order to make this part standalone without referring to part 1.

Language resource management — Feature structures —

Part 2:
Feature system declaration

1 Scope

This part of ISO 24610 provides a format to represent, store or exchange feature structures in natural language applications, for both annotation and production of linguistic data. It is ultimately designed to provide a computer format to define a type hierarchy and to declare the constraints that bear on a set of feature specifications and operations on feature structures, thus offering means to check the conformance of each feature structure with regards to a reference specification. Feature structures are an essential part of many linguistic formalisms as well as an underlying mechanism for representing the information consumed or produced by and for language engineering applications.

A feature system declaration (FSD) is an auxiliary file used in conjunction with a certain type of text that makes use of fs (that is, feature structure) elements. The FSD serves four purposes.

— It provides an encoding by which types and their subtyping and inheritance relationships can be introduced and defined, thus laying the basis for constructing a feature system.

— It provides a mechanism by which the encoder can list all of the feature names and feature values and give a prose description as to what each represents.

— It provides a mechanism by which type constraints can be declared, against which typed feature structures are validated relative to a given theory stated in typed feature logic. These constraints may involve constraints on the range of a feature's value, constraints on which features are permitted within certain types of feature structures, or constraints that prevent the co-occurrence of certain feature-value pairs. The source of these constraints is normally the empirical domain being modelled.

— It provides a mechanism by which the encoder can define the intended interpretation of underspecified feature structures. This involves defining default values (whether literal or computed) for missing features.

The scheme described in this part of ISO 24610 may be used to document any feature system, but is primarily intended for use with the typed feature structure representation defined in ISO 24610-1. The feature structure representations of ISO 24610-1 specify data structures that are subject to the typing conventions and constraints specified using ISO 24610-2. The feature structure representations of ISO 24610-1 are also used within some of the elements defined in ISO 24610-2.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 24610-1:2006, *Language resource management — Feature structures — Part 1: Feature structure representation*

ISO/IEC 19757-2, *Information technology — Document Schema Definition Language (DSDL) — Part 2: Regular-grammar-based validation — RELAX NG*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO 19757-2 and the following apply.

3.1
admissibility constraint
feature admissibility constraint
specification of a set of **admissible features** (3.2) and **admissible feature values** (3.3) associated with a specific **type** (3.24)

3.2
admissible feature
appropriate feature
feature which any **feature structure** (3.14) of a given **type** (3.24) may bear a **value** (3.17) for

NOTE This term is often interpreted elsewhere to mean obligatory, i.e. feature structures of the given type must bear a value for every admissible feature. This term does not imply that the feature is obligatory here.

3.3
admissible feature value
admissible value
value restriction
range restriction
value (3.17) that the value of an **admissible feature** (3.2) must be subsumed by in **feature structures** (3.14) of a given **type** (3.24)

3.4
atomic type
user-defined **type** (3.24) with no **admissible features** (3.2) declared or inherited

3.5
bag
multiset
triple of an integer n, a set S and a function that maps the integers in the range, 1 to n, to elements of S

NOTE A bag is halfway between a set (in that its elements are unordered) and a list (in that particular elements can occur more than once).

3.6
built-in
non-user-defined element that may appear in place of a **feature structure** (3.14), for example, as a **feature value** (3.17)

NOTE Built-ins can be atomic or complex. The atomic built-ins are numeric, string, symbol and binary. The complex built-ins are **collections** (3.7) and applications of the operators, i.e. alternation, negation and merge (5.2.4).

3.7
collection
feature value (3.17) consisting of potentially many values, organized as a list, set or **bag** (3.5)

3.8
constraint
unit of specification that identifies some collection of **feature structures** (3.14) as invalid

NOTE 1 All constraints are implicational in their syntactic form, although some are distinguished as admissibility constraints. See **validity** (3.31) and 5.4. All feature structures not explicitly excluded as invalid are considered to be valid.

NOTE 2 A feature structure that has not been so identified by any of the constraints in a feature system is considered to be valid.

3.9
default value
value (3.17) otherwise assigned to a **feature** (3.12) when one is not specified

EXAMPLE Masculine is the default value of the grammatical gender in Dutch.

NOTE A feature structure may not bear a feature without a corresponding value.

3.10
empty feature structure
feature structure (3.14) that contains no information

NOTE An empty feature structure subsumes all other feature structures.

3.11
extension
converse of **subsumption** (3.21)

NOTE A feature structure F extends G if and only if G subsumes F.

3.12
feature
property or aspect of an entity that is formally represented as a function mapping the entity to a corresponding **value** (3.17)

3.13
feature specification
pairing of a **feature** (3.12) with a **value** (3.17) in a feature structure description

3.14
feature structure
record structure that associates one **value** (3.17) to each of a collection of features

NOTE 1 Each value is either a feature structure or a simpler **built-in** (3.6) such as a string.

NOTE 2 Feature structures are partially ordered. The minimal feature structures in this ordering are the empty feature structures.

3.15
feature system
type hierarchy (3.26) in which each **type** (3.24) has been associated with a collection of **admissibility constraints** (3.1) and **implicational constraints** (3.18)

NOTE cf. **type declaration** (3.25)

3.16
feature system declaration
FSD
specification of a particular **feature system** (3.15)

3.17
feature value
value
entity or aggregation of entities that characterize some property or aspect of another entity

3.18
implicational constraint
constraint of the form, "if G, then H," where G and H are **feature structures** (3.14)

NOTE This identifies any feature structure F as invalid for which G subsumes F, and yet F and H have no valid extension in common. See **subsumption** (3.21) and 8.5. Often used to refer to implicational constraints that are not also admissibility constraints.

3.19
interpretation
minimally informative (or equivalently, most general) **extension** (3.11) of a **feature structure** (3.14) that is consistent with a set of constraints declared by an **FSD** (3.16)

3.20
partial order
partially ordered set
set S equipped with a relation $\leq$ over $S \times S$ that is (1) reflexive (for all $s \in S$, $s \leq s$), (2) anti-symmetric (for all p, $q \in S$, if $p \leq q$ and $q \leq p$, then $p = q$), and (3) transitive (for all p, q, $r \in S$, if $p \leq q$ and $q \leq r$, then $p \leq r$)

NOTE The set of integers Z is partially ordered, but it has an additional property: for every p, $q \in Z$, either $p \leq q$ or $q \leq p$. Not all partial orders have this property. The taxonomical classification of organisms into phyla, genera and species, for example, is a partial order that does not. Type hierarchies may not necessarily. The typed feature structures of a feature system do not, unless (a) their type hierarchy does, and (b) either the type hierarchy has exactly one type, or every y type is constrained to have exactly one appropriate feature.

3.21
subsumption
property that holds between two feature structures, G and F, such that G is said to subsume F if and only if F carries all of the information with it that G does

NOTE A formal definition is provided in 5.6.

3.22
subtype
type (3.24) to which another type confers its constraints and appropriate features

3.23
supertype
base type
type (3.24) from which another type inherits constraints and appropriate features

NOTE s is a subtype of t iff t is a supertype of s. Every type is a subtype and supertype of itself.

3.24
semantic type
type
referring expression that distinguishes a collection of **feature structures** (3.14) as an identifiable and conceptually significant class

NOTE As implied by the name *semantic type*, types in this part of ISO 24610 do not serve to distinguish feature structures or their specifications syntactically.

3.25
type declaration
structure that declares the **supertypes** (3.23), **admissible features** (3.2), **admissible feature values** (3.3), **admissibility constraints** (3.1) and **implicational constraints** (3.18) for a given **type** (3.24)

NOTE The constraints on a type in the resulting feature system are those that have been declared in its declaration, in addition to those that it has inherited from its supertypes.

3.26
type hierarchy
partial order (3.20) over a set of **types** (3.24)

NOTE See ISO 24610-1:2006, Annex C, *Type inheritance hierarchies*.

3.27
typed feature structure
TFS
feature structure (3.14) that bears a **type** (3.24)

3.28
typing
assignment of a semantic **type** (3.24) to a **built-in** (3.6) or **feature structure** (3.14), either atomic or complex

NOTE Semantic types in feature systems are partially ordered, with multiple inheritance.

3.29
underspecification
provision of partial information about a **value** (3.17)

NOTE An underspecification generally subsumes one of a range of candidate values that could be resolved to a single value through subsequent constraint resolution. See **subsumption** (3.21).

3.30
well-formedness
syntactic conformity of a **feature structure** (3.14) representation to ISO 24610-1

3.31
validity
conformity of a **typed feature structure** (3.27) to the **constraints** (3.8) of a particular **feature system** (3.15)

NOTE See Clause 6.

4 Overall structure

The main part of the document consists of four clauses: Clauses 5, 6, 7 and 8.

— Clause 5, *Basic concepts*, reviews the definition of typed feature structures and the notions of atomic and complex types, collections and other operators that may appear in feature values. It then describes the notions of type inheritance hierarchies, type constraints, default values and underspecification that are essential to the construction of feature systems.

— Clause 6, *Defining well-formedness versus validity*, discusses the conditions of well-formedness and validity.

— Clause 7, *A feature system for a grammar*, illustrates how to define types with a type hierarchy and type constraints which declare what features and values are admissible for specific types.

— Finally, Clause 8, *Declaration of a feature system*, discusses how a feature system can be declared and developed into a validator.

The main part of the document is followed by two annexes: Annex A contains the XML schema for this part of ISO 24610; Annex B contains a complete example.

5 Basic concepts

5.1 Typed feature structures reviewed

Typed feature structures (TFSs) are introduced as basic records for language resource management.

For more information, refer to ISO 24610-1:2006, 4.7, *Typed feature structure,* and Annex C, *Type inheritance hierarchies*.

Here, a TFS is formally defined as a tuple over a finite set **Feat** of features, a collection X of non-feature-structure elements, and a type hierarchy **Type**, ≤, where **Type** is a finite set of types and ≤ is a subtyping relation over **Type**.

A feature structure is a tuple <Q, γ, θ, δ>, in which

a) Q is a set of nodes,

b) γ ∈ Q is the root node of the feature structure,

c) θ : Q → **Type** is a partial typing function, and

d) δ : **Feat** × Q → Q ∪ X is a partial feature value function,

such that, for all $q \in Q$, there exists a path of features $F_1, ..., F_n$ such that δ[F_n, ... δ(F_1, γ) ...] = q.

<fs> elements denote nodes. This definition deviates from the standard one used in linguistics and theoretical computer science in that (1) typing is partial, not total, i.e. not all feature structures have types, and (2) feature values might not be feature structures, but instead be drawn from a collection denoted by other XML elements such as string, numeric, symbol, and binary (the X above). Note that nodes are typed, but features themselves are not.

The following XML representation of a feature structure is considered well-formed, where the attribute type is assigned to each of the two <fs> elements.

EXAMPLE Typed feature structure:

```
<fs type="word">
  <f name="orth">
    <string>had</string>
  </f>
  <f name="morphoSyntax">
    <fs type="verb">
      <f name="tense">
        <symbol value="past"/>
      </f>
      <f name="auxiliary">
        <binary value="false"/>
      </f>
    </fs>
  </f>
</fs>
```

The feature name ORTH above stands for orthography, the conventional written form of a word or phrase.

This XML representation shows how the morpho-syntactic features of an English word “had” are specified as a past-tensed and non-auxiliary verb.

In the alternative, “matrix” or “AVM” notation, type names are conventionally in the lower-case, sometimes italicized or in the text type font, feature names in the upper-case, and strings in quotes. Binary values are

indicated with + or –. These conventions are followed in this document, too. The above feature structure would be depicted in matrix notation as shown in Figure 1.

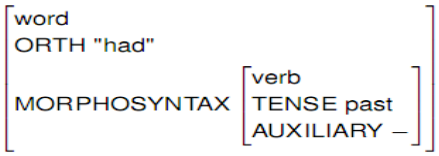

Figure 1 — Matrix notation

5.2 Types

5.2.1 Atomic types

Alongside the built-ins (<symbol>, <string>, <numeric> and <binary>), it is possible for a feature structure to have a type but no features. These are called simple or atomic feature structures, and types that allow for no features in their feature system declaration (FSD) are called atomic types.

There is, as a result, always the possibility of declaring new atomic types and using these instead of the above-mentioned built-ins to specify simple values. The above feature structure, for example, could have instead been rendered as follows, assuming the extra types had, past and false were declared in an FSD.

EXAMPLE Typed feature structure: alternative formulation

```
<fs type="word">
 <f name="orth">
  <fs type="had"/>
 </f>
 <f name="morphoSyntax">
  <fs type="verb">
   <f name="tense">
    <fs type="past"/>
   </f>
   <f name="auxiliary">
    <binary value="false"/>
   </f>
  </fs>
 </f>
</fs>
```

There is a difference also noticed between the two classes of built-ins: <string> on the one hand, and <symbol>, <binary> and <numeric> on the other. Any kind of string is permissible as the content of the <string> element, whereas a very restricted set of values is permissible in <symbol>, <binary> and <numeric> elements. To reflect this difference, members of the latter class specify their values using the attribute *value*. The type <binary>, for instance, is associated with four values: true, false, plus (equivalent to true) and minus (equivalent to false).

NOTE ISO 24610-1:2006 introduced the type *binary*, but the W3C's XML schema (2001) names it *boolean*.

It is the duty of the encoder to choose between atomic-type encodings and built-in encodings consistently. This part of ISO 24610 does not regard one as identical or even consistent with the other.

5.2.2 Complex types

Types that are not atomic are called complex. These include all of the types declared by the encoder in an FSD that declare or inherit admissible features. A feature is only admissible to a type if feature structures of that type are permitted by the FSD to have values for that feature. This does not mean that well-formed feature structures cannot arbitrarily associate types with feature structures regardless of their featural content – they can. But only those feature structures that use only admissible features to their type, as specified by

some FSD, could be validated against that FSD. The distinction between validity and well-formedness is further elaborated upon in Clause 6.

All user-declared types, no matter whether they are atomic or complex, are semantic, i.e. syntactically, they look no different from each other, apart from the value of their type attribute. It is the role of a validator to interpret the real significance of these types through enforcing restrictions on admissibility, restrictions on the possible values that admissible features can have (<vRange>), and other constraints that take the form of logical implications. All of these are specified, for each type, in an FSD.

The built-ins defined by the ISO 24610-1:2006 feature structure representations (FSRs) standard are purely syntactic. They can be used without declaration in an FSD, and they cannot be declared in an FSD. They can appear in value range restrictions, or in implicational constraints, but they cannot have such restrictions (since they have no admissible features) or constraints of their own.

5.2.3 Collections

Not all built-ins are as simple as those mentioned above, however. Some grammatical features such as specifiers (SPR), complements (COMPS) and arguments (ARGS) are considered as having a list of grammatical values, especially in Head-driven Phrase Structure Grammars (Pollard and Sag 1994[10]; Sag, Wasow, Bender 2003[12]). For languages other than English, some of these features may take other kinds of collections, namely sets or multisets, as their value. In a language (e.g. German, Korean or Japanese) that allows a relatively free word order, the feature COMPS may be analysed as taking a set or multiset, instead of a list, of complements. For more general applications, ISO 24610-1:2006 thus introduces sets and multisets as well as lists as built-in ways of assembling complex feature values.

Collections (<vColl>; ISO 24610-1:2006, 5.8, *Collections as complex feature values*) take the organization (org) attribute, with the values “list”, “set” and “bag”. In lists, order and multiplicity of elements matter. In bags, only multiplicity matters (these are often called multisets). In sets, neither order nor multiplicity matter.

For example, the feature ARGS of verbs can be represented by specifying the organization of <vColl> as a list of values, each of which is of type *phrase*.

EXAMPLE List value

```
<fs type="word">
 <f name="orth">
  <string>put</string>
 </f>
 <f name="args">
  <vColl org="list">
   <fs type="phrase">
    <vLabel name="L1"/>
    <f name="nominal">
     <binary value="plus"/>
    </f>
   </fs>
   <fs type="phrase">
    <vLabel name="L2"/>
    <f name="nominal">
     <binary value="plus"/>
    </f>
   </fs>
   <fs type="phrase">
    <vLabel name="L3"/>
    <f name="prepositional">
     <binary value="plus"/>
    </f>
   </fs>
  </vColl>
 </f>
</fs>
```

Some would call the type of this collection *list (phrase)*, but polymorphic lists are not yet supported in this part of ISO 24610. This is equivalent to the following AVM notation, where NP stands for a feature structure of the type *phrase* with a positive NOMINAL feature, namely a noun phrase, and PP, a feature structure of the type *phrase* with a positive PREPOSITIONAL feature, namely a prepositional phrase. The boxed integers are the labels for marking structure sharing as shown in Figue 2.

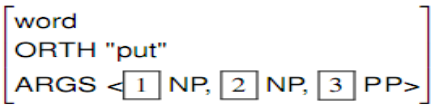

Figure 2 — Marking structure sharing

5.2.4 Operators

The other class of built-ins are operators that take one or more built-ins or feature structures as arguments, but instead of constructing a collection from them, denote a value that is in some other way derived from them.

Alternations (<vAlt>; ISO 24610-1:2006, 5.9.2, *Alternations*) denote one of their arguments' values. A feature structure containing an alternation does not denote multiple feature structures, however. An alternation is a single value that underspecifies which of several possible alternatives it is. Alternations can be regarded as the joins of their arguments in the partial order induced by subsumption (see 5.6).

Negations (<vNeg>; ISO 24610-1:2006, 5.9.3, *Negation*) take a single argument, and denote a value which is not its argument. A negation is equivalent to an alternation among all values that are inconsistent with its argument. A negation is actually not a logical negation of a value, but rather the complement of that value in the full Boolean lattice that contains the partial order induced by subsumption.

A **merge** (<vMerge>; ISO 24610-1:2006, 5.9.4, *Collection of values*) denotes the concatenation or union of several values and/or collections of values, according to how its *org* attribute is set. *org* takes the same values, with the same meanings, as in <vColl>.

5.3 Type inheritance hierarchies

The type hierarchy <Type, ≤> is discussed in great length in ISO 24610-1:2006, Annex C *Type inheritance hierarchies*. This structure is normally depicted as a directed acyclic graph with a unique top node. The label of this top node is often top, and represents the most general type, the type that is consistent with all typed feature structures. Subtypes are connected to, and appear below, their supertypes. The most specific types appear at the bottom of the graph. These are mutually incompatible with each other, which is generally understood implicitly, or on occasion, depicted by another special type, bottom as the unique bottom-most element. Bottom is not used in this part of ISO 24610.

Figure 3 provides an example that depicts a part of the natural world:

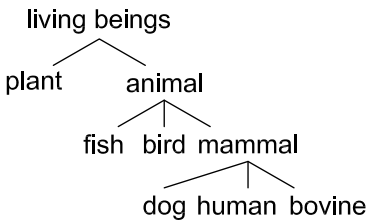

Figure 3 — Type hierarchy for living beings

According to this picture, living beings consist of plants and animals. Animals are subclassified into fish, birds and mammals. Dogs, humans and bovines (oxen, cows, bulls) belong to the class of mammals.

Type hierarchies are not always trees; they may have two or more branches meeting at a single node. When this happens, it means that a type has multiple supertypes, and properties multiply inherited from all of them. Figure 4 provides is an example of this.

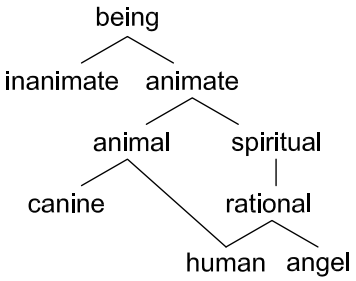

Figure 4 — Medieval hierarchy of beings

Here, the type *human* has two parent types, animal and rational. Hence, a human is viewed as an animal like a dog, but also a spiritual and rational being like an angel. A human thus shares some properties with both dogs and angels.

All these types are partially ordered by a subtyping relation, ≤, over types. A type τ is a subtype of type σ if and only if σ is more general than τ, i.e. if the set of feature structures of type σ contains the set of feature structures of type τ. Since the type *animate* is more general than the type *animal* in the above example, all animals are asserted to be animate. A type σ is said to be a supertype of a type τ if and only if τ is a subtype of σ. The immediate supertypes of a type are often called its parents.

A subtype inherits all of the properties from its supertype. The type *human*, for instance, inherits all the properties from its supertypes (*being*, *animate*, *animal*, *spiritual* and *rational*).

Here is a linguistic example, modified from Grammar 2 of Copestake (2002)[2] is shown in Figure 5.

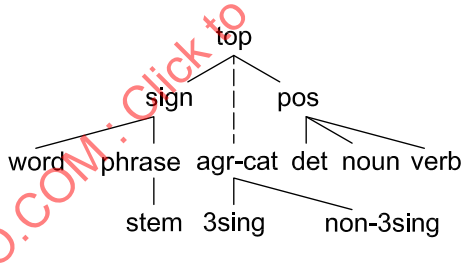

Figure 5 — Type hierarchy for a simple grammar top

The type hierarchy has a unique top element. It is the most general type with no parents or immediate supertypes. *Top* is only a subtype of itself.

Each type has a name and every type, except for the top-most type, has exactly one parent. The type *top* has four immediate subtypes. *phrase* and *det* are incomparable – neither is a subtype or supertype of the other.

Depending on the complexity of a grammar, the type hierarchy can be very complex. Some portions of the hierarchy may be universal to all languages, while others are very language-specific. The agreement type *agr-cat* in English, for example, has only two immediate subtypes: *3sing* and *non-3sing* (e.g. “sings” versus “sing”).

The type *det* stands for a determiner such as “the” or “a”; *3sing* stands for 3rd person singular, and *non-3sing* stands for agreement categories other than *3sing*.

This distinction is the one that is apparent in English verb agreement.

5.4 Type constraints

The type hierarchy is the skeleton on which the rest of the grammar grows. The rest of the grammar takes the form of constraints over feature structures of these user-defined types. These constraints are at least of the following three kinds: (1) implicational constraints, (2) constraints on admissible features, and (3) constraints on admissible feature values. Actually, all of them can be thought of implicationally, e.g.

— if a feature structure is of type *verb*, then it may have the feature AUXiliary,

— if a feature structure is of type *verb*, then it may have the feature INVerted,

— if a feature structure is of type *verb*, then its AUX value must be “binary”,

— if a feature structure is of type *verb*, then its INV value must be “binary”,

— if a feature structure is of type *verb* and its AUX value is negative, then its INV value must be “negative”.

The first two of these are feature admissibility constraints. They tell us that a particular feature can be used in feature structures of a particular type. The second two are constraints on admissible feature values, sometimes called “value restrictions” or “range restrictions”. They tell us what kind of value a particular feature must take when it occurs in a feature structure of some given type. The last of these is of a more general form; however, this kind of constraint says that whenever a feature structure takes some particular form (determined by types, feature values, etc.), it must satisfy some other criteria (again stated in terms of types, feature values, etc.). This last form of constraint is generally what is meant by the phrase implicational constraint. Each of these three forms has a different syntax in an FSD. The above constraints on verb would be encoded as follows.

EXAMPLE Constraint on the type *verb*

```
<fsDecl type="verb">
 <fDecl name="aux">
  <vRange>
   <binary/>
  </vRange>
 </fDecl>
 <fDecl name="inv">
  <vRange>
   <binary/>
  </vRange>
 </fDecl>
 <fsConstraints>
  <cond>
   <fs>
    <f name="aux">
     <binary value="false"/>
    </f>
   </fs>
   <then/>
   <fs>
    <f name="inv">
     <binary value="false"/>
    </f>
   </fs>
  </cond>
 </fsConstraints>
</fsDecl>
```

The first two kinds are specified together inside an <fDecl> element, the second being the <vRange> portion of that declaration, whereas the third is specified as an if-then conditional (<cond>).

5.5 Optional (default) values and underspecification

In a feature structure, some features must be specified and others need not be. In French, for example, the specification of the features NUMBER and GENDER is obligatory for nouns and adjectives. In English, the feature NUMBER must be specified for each noun, but the specification of the feature GENDER is optional. It is obligatory for the third person singular pronouns, “he”, “she” and “it”.

Nevertheless, there are cases in which some obligatory features are not specified. In those cases, there are two possibilities: (1) if a default value has been defined, then it is understood to be its value; and (2) if not, then the value of the feature is inferred from the feature's range restriction.

English mass nouns such as “water” and “air” are uncountable and singular by default. Hence, their NUMBER feature need not be specified, although the feature NUMBER is obligatory. Some countable nouns such as “sheep” can be either singular or plural. When the NUMBER is not specified, its value is understood to be some more general type such as *number*, which is a supertype of all of the admissible feature values.

Grammatical descriptions are often underspecified in order to capture generalizations. In English, for instance, verbs are subclassified into the number of complements that they require. Intransitive verbs (“smile”, “bark”) take only a subject, transitive verbs (“love”, “attack”) take a subject and a direct object, and ditransitive verbs (“give”, “put”) take a subject, direct object and indirect object. Many of the grammatical phenomena, however, do not refer to one of these specific subclasses. Examples include subject-verb agreement (“The dog barks” versus ill-formed “the dog bark”) or subject-verb inversion (“Does the dog bark?” versus ill-formed “Do the dog attacks Jane?”). Since the specification of this feature is irrelevant in describing these grammatical phenomena, it is left underspecified.

Here is another example for underspecification. The analysis of a sentence like “The sheep attacked Jane” may be underspecified with respect to the NUMBER value of “sheep”. Only if necessary is its lexical ambiguity displayed.

Default values are specified in FSDs with <vDefault>, as explained in 8.4, and can be referenced in FSRs with the <default> element (ISO 24610-1:2006, 5.10, *Default values*).

5.6 Subsumption

A feature structure F subsumes another feature structure G ($F \subseteq G$) if and only if G contains all of the information that F does. “Information” is delivered by a feature structure in two ways: typing and path equality. When one views feature structures as a pair consisting of an equivalence relation on paths (≡), and a partial typing function on paths (Θ), then formally $< \equiv_F, \Theta_F > \subseteq < \equiv_G, \Theta_G >$ if and only if $\equiv_F \subseteq \equiv_G$ and for all $\pi \in \text{Paths}_F \cap \text{Paths}_G$, if $\Theta_F(\pi)$ is defined, then $\Theta_G(\pi)$ is defined and $\Theta_G(\pi)$ is a subtype of $\Theta_F(\pi)$. When $F \subseteq G$, we say that G extends F.

The view of typed feature structures taken here is still more general than is often the case in either the linguistics literature or the formal literature on typed feature logic, because of the presence of symbols, strings, numbers and other feature values than <fs> elements. With respect to extensions and subsumption, strings, symbols, numbers and Booleans (binary values) behave as though they were types with no admissible features that are discretely ordered alongside, but not connected to, the rest of the type inheritance hierarchy, i.e. they have no subtype relationships with any type but themselves. Feature structures of these “types” are only subsumed by themselves and the most general untyped feature structure, <fs/>, and they have no extensions other than themselves. Some caution must be exercised, however, with respect to determining subsumption within this extended view of typed feature structures, because re-entrancies may still exist or not exist between identical-looking symbols, strings, numbers, etc. The more extensional view of identity that usually accompanies these other entities is inconsistent with the view of identity that the logic of typed feature structures takes with respect to feature structures over its own types. It is the latter that this part of ISO 24610 uses and applies to both feature structures and these other entities, when they occur within feature structures.

Alternations are also often excluded from more formal work on typed feature logic, but can be thought of as joins of their respective typed feature structure arguments in the partial order of typed feature structures induced by subsumption. The negation of a value can similarly be thought of as the join of every structure that

is inconsistent with that value under unification. Collections actually depend on the organization. Lists appear in the subsumption partial order as though they were encoded as typed feature structures using this FSD.

EXAMPLE Sample FSD

```
<fsDecl type="list" baseTypes="top"/>
<fsDecl type="e-list" baseTypes="list">
 <fsDescr>Empty lists</fsDescr>
</fsDecl>
<fsDecl type="ne-list" baseTypes="list">
 <fsDescr>Non-empty lists</fsDescr>
 <fDecl name="first"/>
 <fDecl name="rest">
  <vRange>
   <fs type="list"/>
  </vRange>
 </fDecl>
</fsDecl>
```

One bag (multiset) B_1 subsumes another bag B_2 if and only if there exists a total surjection σ between the elements of the two bags such that, for all b_1 in the domain of B_1 with multiplicity $\mu_1(b_1)$, and all b_2 in the domain of B_2 with multiplicity $\mu_2(b_2)$:

1) $b_1 \subseteq \sigma(b_1)$,

2) $\mu_2(b_2) = \sum_{b_1 : \sigma(b_1) = b_2} \mu_1(b_1)$,

and σ can be extended to a total function, σ*, between the substructures of the elements of the two bags, such that, for all substructures, c, of the elements of B_1:

3) $\sigma^*(c) = \sigma(c)$ if c is an element of B_1, and

4) $\sigma^* [\delta(F, c)] = \delta[F, \sigma^*(c)]$, for every $F \in$ **Feat** such that $\delta(F, c)$ is defined.

One set S_1 likewise subsumes another set S_2 if and only if conditions 1), 3) and 4) above apply. This means, for example, that the two-element set $\{F_1, F_2\}$ subsumes the one-element set $\{G_1\}$ if both $F_1 \subseteq G_1$ and $F_2 \subseteq G_1$. This partially ordered interpretation of sets is called the Pollard-Moshier set theory, and it is one of the most commonly used theories in typed feature logic.

In addition, a bag subsumes any list that is a permutation of its elements. A set subsumes a bag if the domain of the bag is the set, i.e. all and only the elements of the set appear in the bag one or more times.

A combination of collections (<vMerge>) occupies the same position in the subsumption partial order as the result of the concatenation or union that it specifies, with the organization it specifies, would have.

The reflexive and transitive closure of all of these conditions produces the subsumption relation assumed by this part of ISO 24610.

6 Defining well-formedness versus validity

6.1 Overview

6.1.1 General

This clause distinguishes the use of the concepts of well-formedness and validity, as they pertain to feature structure representations and feature systems. In linguistic theory, even linguistic theories ostensibly based on typed feature logic, they are often used synonymously or in ways that diverge from their conventional meanings in either formal logic or XML. Their usage in formal logic and XML also differs. Before formulating these two concepts, a brief survey is made of how they have been treated in these two areas.

6.1.2 Formal logics

In formal logics, the concepts of well-formedness and validity are clearly distinguished. Well-formedness is a syntactic concept, whereas validity is a semantic concept. A string of symbols in a logic is well-formed if it is defined by a set of its formation rules. In first-order logic, for instance, the string $\forall x\,[H(x) \rightarrow [G(x) \rightarrow H(x)]]$ is considered as a well-formed formula, where $\forall$ is the universal quantifier, x is an individual variable, the arrow $\rightarrow$ is a binary sentential operator, G and H are unary predicate symbols, and all the brackets properly match. On the other hand, $\forall x$ by itself is ill-formed because the syntactic formation rule for quantifiers requires each quantifier with a variable to be followed by a well-formed formula. The syntactic formation rules thus delineate the set of well-formed formulas out of the set of arbitrary strings.

The semantics of first-order logic then interprets these well-formed formulas by evaluating their truth values. Since ordinary first-order logic is a bivalent logic, every formula including atomic formulas is either true or false with respect to some interpretation (or model) and possibly also with respect to an assignment of values to variables in the case of so-called open formulas like $G(x)$ and $H(x)$. A formula $G(x)$ is true with respect to some model and some assignment if and only if the value assigned to the variable x belongs to the denotation set of G in that model. Suppose x is Jane and G refers to girls. Then, $G(x)$ is true, assuming that Jane is a girl. But the formula $\forall x[H(x) \rightarrow [G(x) \rightarrow H(x)]]$ is always true with respect to any model or any variable assignment, for it is the form of a tautology $[p \rightarrow [q \rightarrow p]]$ in propositional logic. Such a formula is called valid. In general, a well-formed formula is said to be valid if it is true under every interpretation/model. One of the semantic tasks in logics is thus to delineate all and only the valid formulas out of the total set of well-formed formulas.

6.1.3 XML

In XML again, a clear distinction is made between these two concepts. XML documents can be either well-formed or ill-formed (broken) and then well-formed documents can be either valid or non-valid (broken). Just like most other markup languages, a well-formed XML document must follow several rules like one-root element, proper nesting, and well-formed entities. The criteria for the validity of XML documents are slightly more difficult to set up. First, valid documents must be well-formed and then validated against all the constraints (rules) set forth in a document grammar such as a Document Type Definition (DTD), internal or external, an XML schema, a RELAX NG or some other format. Non-valid XML documents are also called broken.

6.2 ISO 24610

6.2.1 Definitions

In this International Standard, well-formedness and validity are treated as clearly distinct, and in a fashion that can generally be understood as analogous to XML, with the role of DTDs being played by the FSDs defined in this International Standard.

— A feature structure representation is well-formed if and only if it conforms to the definitions of feature structure declarations and typing, as specified in ISO 24610-1:2006 or its updated version.

— Corollary: every feature structure representation in XML must follow the well-formedness conditions of XML documents like having a single root condition, proper nesting, and well-formed entities.

— A feature structure representation in XML is valid if and only if it is well-formed and also conforms to the feature system declared (in a DTD, an XML schema or some other format) for a particular application that uses typed feature structures.

A well-formed feature structure representation is typed relative to a (in the XML sense) well-formed FSD if and only if every <fs> it contains (including itself, if it is an <fs> element) bears a value for the type attribute that is declared as such in some <fsDecl> of the FSD. Every typed feature structure representation uniquely denotes a typed feature structure. In this International Standard, we speak of typed feature structures and typed feature structure representations interchangeably. Not all feature structures in ISO 24610-1:2006 are typed. Validity and validation with respect to an FSD, however, only make sense for typed feature structure representations.

A typed feature structure can be invalid in several different ways. To be valid, it must, first of all, be well-formed, satisfying both the well-formedness conditions of XML and the definitions of feature structures and typing. Then the values of its features must fall within the admissible feature value ranges declared. Finally, it must satisfy the type constraints imposed on it as well as the constraints inherited from its base types.

Unlike in XML, typed feature structures are partially ordered by the subsumption relation (see 5.6). It simply makes no sense to refer to subsumption of or by an ill-formed feature structure representation, but it does make sense to speak of subsumption of or by non-valid feature structure representations so long as they are typed. Valid typed feature structures can subsume non-valid ones, and vice versa, so validity and subsumption cannot be used to make inferences about each other, but it is comparatively rare that we are ever interested in determining the simple validity of a typed feature structure. Instead, the process of validation almost always involves the search for a valid extension (3.11) of a typed feature structure. Every valid typed feature structure trivially has at least one valid extension, namely itself. Some non-valid typed feature structures have no valid extensions, but the ones that have one have a unique most general (or equivalently, least informative) valid extension. This most general valid extension is used as a proxy for the entire set of valid extensions that a valid or non-valid typed feature structure subsumes.

One often finds other kinds of typed feature structures distinguished in the literature. Most notably, totally well-typed feature structures are typed feature structures in which every obligatory feature takes a value that respects the feature's value range (<vRange>) restrictions – this is tantamount to validity without consideration for the <cond> or <bicond> constraints of the FSD. In addition, what are here called *feature structure representations* accord in many respects more closely with what computational linguists would term *feature structure descriptions* than with feature structures. The description language used in most linguistic applications [of typed feature structures, the vast majority of them being in the Head-driven Phrase Structure Grammar (HPSG) framework: Pollard and Sag, 1994[10]] is conservative enough to easily embed into ISO 24610-1:2006 feature structure representations. However, there are ISO 24610-1:2006 FSRs for which the correspondence to a single, equivalent description is, at best, remote, owing to their dependence on the FSD relative to which such an equivalence could be proved.

6.2.2 Review of the syntax of typed feature structures in XML

6.2.2.1 Overview

The overview of the syntax of typed feature structure representations is provided here by introducing the relevant element names and their patterns.

6.2.2.2 Introducing names

a)	element names for feature structures and features:	fs, f
b)	element names for feature values:	(fs), string, symbol, binary, numeric, vLabel
c)	attribute names for elements:	name, type, org, value
d)	element names as collection constructors:	vColl
e)	element names as operators:	vAlt, vNot, vMerge, default

6.2.2.3 Basic pattern

```
<fs type="Type">
 <f name="featureName">
  <fs type="featureValueType">VALUE</fs>
 </f>
</fs>
```

6.2.2.4 Feature value patterns

a) for atomic feature value types

```
<fs type="atomicType"/>
```

b) for feature structures as values

```
<fs type="featureValueType">VALUE</fs>
```

c) collections

```
<vColl org="collectionType">
 <fs type="Member1Type">VALUE1</fs>
 <fs type="Member2Type">VALUE2</fs>                                    ...
</vColl>
```

d) alternation

```
<vAlt>
 <fs type="Disjunct1Type">VALUE1</fs>
 <fs type="Disjunct2Type">VALUE2</fs>                                 ...
</vAlt>
```

e) negation (complement)

```
<vNot>
 <fs type="NegatedValue">VALUE</fs>
</vNot>
```

or

```
<vNot>
 <vAlt>
  <fs type="NegatedValue1">VALUE1</fs>
  <fs type="NegatedValue2">VALUE2</fs>
 </vAlt>
</vNot>
```

EXAMPLE

```
<fs type="pos">
 <f name="agr">
  <fs type="agr-cat">
   <f name="per">
    <vNot>
     <fs type="3rd"/>
    </vNot>
   </f>
   <f name="num">
    <vNot>
     <fs type="singular"/>
    </vNot>
   </f>
  </fs>
 </f>
</fs>
```

6.2.3 Illustrations for well-formedness

By definition, a feature structure is a partial function from features to values. Hence, the representation fs/ is allowed (representing the empty feature structure), but a representation that specifies features without values is not allowed.

EXAMPLE 1 Well-formedness

a) <**fs type**="top"/>

b) <**fs type**="TYPE">
<**f name**="FEATURE"/>
<!-- WRONG -->
</**fs**>

Here, a) is well-formed, while b) is ill-formed.

A type is assigned to each feature structure or feature value, but not to a feature. Hence, an element named "f" may not have an attribute named "type".

EXAMPLE 2 Well-formedness

a) <**fs**/>

b) <**fs type**="top"/>

c) <**fs type**="TYPE1">
<**f name**="FEATURE" **type**="TYPE2">
<**fs type**="top"/>
</**f**>
<!-- WRONG -->
</**fs**>

d) <**fs type**="TYPE1">
<**f name**="FEATURE">
<**fs type**="TYPE2"/>
</**f**>
</**fs**>

Here,

a) is well-formed – this is the most general untyped feature structure representation;

b) is also well-formed; *top* is conventionally considered to be the most general type, which would make this the most general type of feature structure representation, although the standard does not require this and *top* is not built-in – *top* must still be declared if it is to be used;

c) is not well-formed because a type is assigned to a feature;

d) is well-formed because a feature value may be typed.

6.2.4 Illustration for validity

6.2.4.1 Conditions

The conditions of validity depend on a particular feature system that consists of type constraints. For illustration, consider the specification of type constraints in 6.2.4.3. The following typed feature structure is valid with respect to this FSD.

6.2.4.2 Illustration for validity

```
<fs type="word">
 <f name="orth">
  <string>Mia</string>
 </f>
 <f name="head">
  <fs type="noun">
   <f name="agr">
    <fs type="agr-cat">
     <f name="person">
      <fs type="3rd"/>
     </f>
     <f name="number">
      <fs type="singular"/>
     </f>
    </fs>
   </f>
  </fs>
 </f>
 <f name="spr">
  <vColl org="list"/>
 </f>
 <f name="comps">
  <vColl org="list"/>
 </f>
</fs>
```

The corresponding AVM is shown in Figure 6.

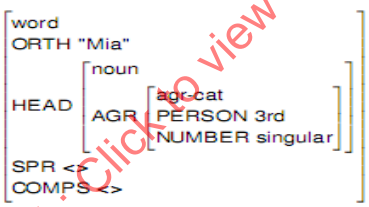

Figure 6 — Corresponding AVM

NOTE Again, the element names *orth*, *head*, *agr*, *spr* and *comps* in the above XML representation each correspond to *ORTH*, *HEAD*, *AGR*, *SPR* and *COMPS* in the AVM representation.

The example this subclause represents the valid typed feature structure in Figure 6. This TFS is valid because it satisfies all the constraints for the type *word*. Below is the most general valid typed feature structure of that type. It subsumes this TFS or Figure 6.

6.2.4.3 Overall constraint for the type *word*

```
<fs type="word">
 <f name="orth">
  <fsval kind="string"/>
 </f>
 <f name="head">
  <fs type="pos"/>
 </f>
 <f name="specifier">
  <vColl org="list"/>
 </f>
 <f name="complements">
  <vColl org="list"/>
 </f>
</fs>
```

The corresponding AVM is shown in Figure 7.

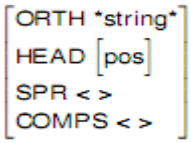

Figure 7 — Corresponding AVM

It can be determined feature-by-feature how this feature structure subsumes the TFS given in 6.2.4.2. In this TFS, a string “Mia” is given for ORTH, the type *noun* is a subtype of *pos* for HEAD, and the empty list is a kind of list collection for both SPEC and COMPS.

7 A feature system for a grammar

7.1 Overview

Typed feature structures are very widely used for the development of grammars, lexica and other linguistic resources and applications. In a grammar implementation like the English Resource Grammar (ERG; Flickinger, 2002[3]), everything that constitutes a grammar, including type definitions, phrase structure rules, and lexical entries, is represented by feature structures. In this approach, each of these feature structures must be uniquely extensible to a most general feature structure that satisfies an accompanying type system, which consists of a type hierarchy and a set of type constraints. As will be observed in 8.4, some types may also be associated with a collection of admissible features and their admissible feature values, as well as more general implicational constraints. Since feature values can be obligatory (required), optional or default, this distinction should also be marked in a feature system.

These admissibility declarations as well as the individual subtype relationships that constitute the type hierarchy can be, and often are, expressed as typed feature structures, with the context determining that they should be viewed as assertions about validity, and not merely as data. Under suitable syntactic restrictions to the allowable constraints of this International Standard – restrictions that one can observe in operation in the ERG and its derivative fragments – implicational type constraints can also be encoded as object-level feature structures, i.e. feature structures drawn from the same type system as the underlying grammar. Under the ERG's conventions, all type constraints are unidirectional implications (<cond>), in which the antecedent is only a type. In addition, both the lexicon and the phrase structure rules of a grammar can be regarded as object-level type constraints through the use of alternations (ISO 24610-1:2006, 5.9.2, *Alternations*), which are otherwise foreign to the ERG.

Even without assuming the conventions and restrictions of the previous paragraph, however, a meta-level representation as feature structures is always available for every component of a grammar. In such a representation, extra types and features are used to encode, in the case of phrase structure rules, for example, a left-hand- or right-hand-side constituent of the corresponding rule. A head-complement rule that consumes one argument of a phrasal head, for example, could be encoded as shown in Figure 8.

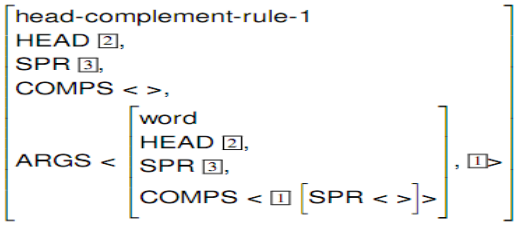

Figure 8 — Head-complement rule 1

Here, an extra type (head-complement rule 1) and feature (ARGS) have been appropriated for representing, respectively, the identity and right-hand-side constituents of this rule. Viewed as an implicational constraint, the extra type can also be viewed as the antecedent, and everything else as a conjunctive consequent. Lists

(expressed with angled brackets as a shorthand), one of the complex collections of ISO 24610-1:2006, 5.8, *Collections as complex feature values*, are also being used here, although they may not have any other role in the object-level type system of the grammar.

In linguistic publications, these encodings may be sufficient, but do not be misled by such informal practices. Under this International Standard, FSDs as defined here, and not merely the <fs> elements of ISO 24610-1:2006, must be used to declare the type system, admissible features, admissible feature value restrictions, and implicational type constraints of a grammar. Meta-level encodings are always possible, but possibility is not at issue. FSDs document the proper intention, i.e. how the information is to be used by an application, in which there may not be a human in the loop to infer it.

A grammar that is consistent with the present standard, therefore, is minimally defined by: (1) a feature system declaration (type hierarchy, admissibility declarations, and type constraints), (2) a lexicon, and (3) a collection of phrase structure rules. (1) must use FSDs. It is recommended that (2) and (3) be encoded through syntactic conventions that exist outside this International Standard, in which their status as lexical items or production rules is also explicitly and unambiguously indicated, but it is also permissible to use FSDs.

7.2 Sample FSDs

7.2.1 General

In this subclause, an illustration of a sample grammar's feature system declaration, modified from Grammar 2 in Copestake (2002)[2], is shown.

7.2.2 Defining types and their hierarchy

Types are defined by specifying their supertype parents. Type hierarchies are not always trees – a subtype can have more than one supertype parent – but there are restrictions on their shape, such as having a unique most general type.

A sample feature system declaration for a type hierarchy is shown in Figure 9.

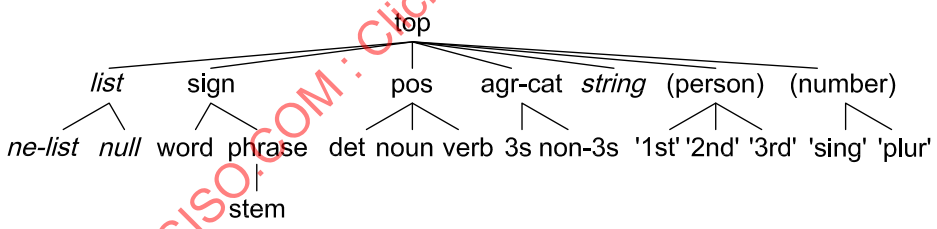

Figure 9 — A simple type hierarchy for English

Each branch in this tree simply declares an instance of a subtype-supertype relationship; *top* or *T* is often used as the name of the type that applies to every feature structure. Italicized types like *list* and *string* are built-ins that would not receive an explicit declaration in a conforming FSD (shown below). Parenthesized types are types that could have been declared, but instead the constraints below use alternations among their subtypes, which are encoded as symbols (shown as single-quoted). Note also that the lists introduced here are non-polymorphic.

The difference between a type system, which is what FSDs actually declare, and a type hierarchy is that type systems specify not only subtyping relationships, but also admissible features, restrictions on the admissible feature values of admissible features, and other constraints on types and their features' values.

7.2.3 Declaring type constraints

In this example, there are no type constraints apart from restrictions on admissible feature values, so <cond> and <bicond> (see 8.5) do not need to be used. We begin by presenting the admissible features and their

admissible feature values informally. Here, each feature structure indicates the admissible features and minimal admissible feature values for its respective type. See Figure 10.

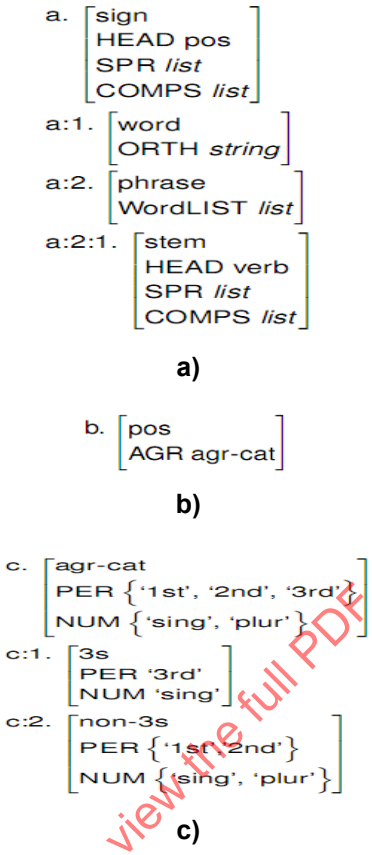

Figure 10 — Admissible features and their admissible feature values

Note that 3s and non-3s refine the restrictions on the admissible feature values of PER and NUM by reducing the size of the alternation that constrains them.

Taken altogether, this type system is encoded as follows.

EXAMPLE Type system

```
<fsDecl type="sign">
 <fsDescr>The fundamental type for linguistic signs</fsDescr>
 <fDecl name="head">
  <fDescr>Indicates the syntactic head of the sign</fDescr>
  <vRange>
   <fs type="pos"/>
  </vRange>
 </fDecl>
 <fDecl name="spr">
  <fDescr>Indicates the specifiers of the sign</fDescr>
  <vRange>
   <vColl org="list"/>
  </vRange>
 </fDecl>
 <fDecl name="comps">
  <fDescr>Indicates the complements of the sign</fDescr>
  <vRange>
   <vColl org="list"/>
  </vRange>
```

```
 </fDecl>
</fsDecl>
<fsDecl type="word" baseTypes="sign">
 <fsDescr>The fundamental type for individual words</fsDescr>
 <fDecl name="orth">
  <fDescr>The orthographic representation for this word</fDescr>
  <vRange>
   <string/>
  </vRange>
 </fDecl>
</fsDecl>
<fsDecl type="phrase" baseTypes="sign">
 <fsDescr>The fundamental type for phrasal signs</fsDescr>
 <fDecl name="wordlist">
  <fDescr>The words contained within this phrase</fDescr>
  <vRange>
   <vColl org="list"/>
  </vRange>
 </fDecl>
</fsDecl>
<fsDecl type="stem" baseTypes="phrase">
 <fsDescr>The verbal stem from which a phrase is formed</fsDescr>
 <fDecl name="head">
  <fDescr>The verbal head that phrases formed from this stem must
    take</fDescr>
  <vRange>
   <fs type="verb"/>
  </vRange>
 </fDecl> copyright ISO 2011 – All rights reserved ISO 24610-2 <fDecl name="spr">
  <fDescr>Indicates the specifiers of the sign</fDescr>
  <vRange>
   <vColl org="list"/>
  </vRange>
 </fDecl>
 <fDecl name="comps">
  <fDescr>Indicates the complements of the sign</fDescr>
  <vRange>
   <vColl org="list"/>
  </vRange>
 </fDecl>
</fsDecl>
<fsDecl type="pos">
 <fsDescr>Parts of speech</fsDescr>
 <fDecl name="agr">
  <fDescr>Agreement information for this part of speech</fDescr>
  <vRange>
   <fs type="agr-cat"/>
  </vRange>
 </fDecl>
</fsDecl>
<fsDecl type="agr-cat">
 <fsDescr>GPSG-style agreement complex</fsDescr>
 <fDecl name="per">
  <fDescr>Person value</fDescr>
  <vRange>
   <vAlt>
    <symbol value="1st"/>
    <symbol value="2nd"/>
    <symbol value="3rd"/>
   </vAlt>
  </vRange>
```

```
 </fDecl>
 <fDecl name="num">
  <fDescr>Number value</fDescr>
  <vRange>
   <vAlt>
    <symbol value="sing"/>
    <symbol value="plur"/>
   </vAlt>
  </vRange>
 </fDecl>
</fsDecl>
<fsDecl type="3s" baseTypes="agr-cat">
 <fsDescr>Third-person singular agreement complex</fsDescr>
 <fDecl name="per">
  <fDescr>Person value</fDescr>
  <vRange>
   <symbol value="3rd"/>
  </vRange>
 </fDecl>
 <fDecl name="num">
  <fDescr>Number value</fDescr>
  <vRange>
   <symbol value="sing"/>
  </vRange>
 </fDecl>
</fsDecl>
<fsDecl type="non-3s" baseTypes="agr-cat">
 <fsDescr>Third-person singular agreement complex</fsDescr>
 <fDecl name="per">
  <fDescr>Person value</fDescr>
  <vRange>
   <vAlt>
    <symbol value="1st"/>
    <symbol value="2nd"/>
   </vAlt>
  </vRange>
 </fDecl>
 <fDecl name="num">
  <fDescr>Number value</fDescr>
  <vRange>
   <vAlt>
    <symbol value="sing"/>
    <symbol value="plur"/>
   </vAlt>
  </vRange>
 </fDecl>
</fsDecl>
```


Note that some features have been redeclared as admissible as subtypes of types for which they have already been declared. This is permissible, although if the multiply inherited feature value restrictions are not unifiable, there will be no valid feature structures of that type; *top* only needs to be defined if it will actually be named as a type in feature structures.

8 Declaration of a feature system

NOTE This clause is a slightly modified version of the *TEI Guidelines P5*, 2005, Section 18.11, *Feature System Declaration*. A fuller discussion of the reasoning behind FSDs and another complete example are provided by “A rationale for the TEI recommendations for feature-structure markup”, by D. Terence Langendoen and Gary F. Simons, in *Computers and the Humanities*, **29**, 1995[7].

8.1 Overview

The Feature System Declaration (FSD) standard is intended for use in conjunction with fs (that is, feature structure) elements that comply with ISO 24610-1:2006, although it may be used to document any feature structure system. Its purposes are stated in Clause 1, Scope.

The FSD serves an important function in documenting precisely what the encoder intended by the system of feature structure markup used in an XML-encoded text. The FSD is also an important resource which standardizes the rules of inference used by software to validate the feature structure markup in a text, and to infer the full interpretation of underspecified feature structures.

The reader should be aware, however, that there are a number of terminological mismatches between the present standard and conventional practice in both formal logic and practical linguistic applications of typed feature structures. In particular, what shall be called an “interpretation” of a feature structure here is not an interpretation in the model-theoretic sense, but is instead a minimally informative (or equivalently, most general) extension (see 5.6) of that feature structure that is consistent with a set of constraints declared by an FSD. In linguistic application, such a system of constraints is the principal means by which the grammar of some natural language is expressed. There is, however, a great deal of disagreement as to what, if any, model-theoretic interpretation feature structures have in such applications. This status of this formal kind of interpretation is not relevant to the present standard. The term *valid* normally appeals to a notion of formal semantics as well, but is merely used here to describe what is in fact a purely syntactic state of well-formedness in the sense defined by the logic of typed feature structures itself, as distinct from and in addition to the “well-formedness” that pertains at the level of this encoding standard (see Clause 6).

The following subclause describes how an XML-encoded text should use header information to make links to any associated FSDs. The third, fourth and fifth subclauses describe the overall structure of an FSD and provide details on how to encode its parts. Annex B offers a full example.

8.2 Linking a text to feature system declarations

In order for application software to use feature system declarations to aid in the automatic interpretation of encoded texts, or even for human readers to find the appropriate declarations which document the feature system used in markup, there must be a formal link from the encoded texts to the declarations. However, the schema that declares the syntax for the feature system should be kept distinct from the feature structure representation schema itself, which is an application of that system.

The association between an FSD and a document using the feature structures it declares is reified in this International Standard in a manner intended to be consistent with its inclusion in the <encodingDesc> block of a document's <teiHeader> (see *the TEI Guidelines P5*, Section 2.3[14]). The fsdDecl element may be used for each distinct feature structure type, as follows (in this International Standard, we use the “compact” variant of the RELAX NG schema language to define such elements):

```
element fsdDecl
{
    att.global.attributes,
    attribute type { data enumerated }?
    attribute url { data.pointer },
    empty
}
```

<fsdDecl> [FSD (feature-system declaration) declaration] identifies the feature system declaration which contains definitions for a particular type of feature structure. In addition to global attributes, type identifies the type of feature structure documented by the FSD; it is expected that this will be the value of the type attribute on at least one feature structure. The value can be any string of characters, but if the value contains whitespace, it must be normalized: no leading or trailing sequences of whitespace characters, nor internal sequences of more than one whitespace character. The type attribute is optional. If better validation is required, the global xml:id attribute may be used to specify the type instead of this attribute, in which case the name must be a valid identifier. If neither is used, then this <fsdDecl> is assumed to identify the FSDs for all types of feature structures used in the encoding.

url supplies a link to the entity containing the feature system declaration. Its value must be an RFC 2396 Uniform Resource Identifier (URI).

There may be multiple *fsdDecl* elements for a given FSD; one for each type of feature structure it defines. For instance, in the following example, the file *Lexicon.fsd* contains an FSD that contains definitions of feature structures for lexical entries (<fs type="entry">) and lexical subentries (<fs type="subentry">). The file *Gazdar.fsd* contains another FSD which contains the definition of a type of feature structure called **GPSG**:

```
<TEI>
 <teiHeader>
  <fileDesc>
<!-- ... -->
  </fileDesc>
  <encodingDesc>
<!-- ... -->
   <fsdDecl type="GPSG" url="Gazdar.fsd"/>
   <fsdDecl type="entry" url="Lexicon.fsd"/>
   <fsdDecl type="subentry" url="Lexicon.fsd"/>
<!-- ... -->
  </encodingDesc>
 </teiHeader>
<!-- The text goes here -->
</TEI>
```

This example shows an <fsdDecl> being given within the <encodingDesc> for each distinct value used as the type of the <fs> elements in the document itself. In this case, for example, the feature system declaration used by feature structures of types entry and subentry is to be found in the entity at the URL **Lexicon.fsd**.

The current standard provides no way of enforcing the uniqueness of type values among fsdDecl elements, nor of requiring that every type value specified on an <fs> element be also declared on an <fsdDecl> element, nor of ensuring that multiple <fsdDecl> do not appear in the same <encodingDesc> both with and without the optional type attribute.

Encoders requiring such constraints (which might have some obvious utility in assisting the consistency and accuracy of tagging) are recommended to develop tools to enforce them, using such mechanisms as Schematron assertions.

FSDs avail themselves of the following elements: *fsd*, *fsDecl*, *fsDescr*, *fDecl*, *fDescr*, *vRange*, *vDefault*, *if*, *then*, *fsConstraints*, *cond*, *bicond* and *iff*.

While the syntax of FSDs has no particular dependencies on any TEI module or ISO 24610-1 FSRs, it should be used in conjunction with the standard tei, header, and core modules, as well as ISO 24610-1.

Broadly speaking, an FSD consists of one or more feature structure declarations (<fsDecl>), one or more feature definitions (<fDecl>), and zero or more feature structure constraints (<cond> and/or <bicond>). Feature definitions and feature structure constraints occur only within the scope of feature structure declarations.

8.3 Overall structure of a feature system declaration

A feature system declaration is encoded as a document of type <fsd>. Apart from its global attributes, it has two parts: an optional header (which provides bibliographic information for the file) and a set of feature structure declarations, each of which defines one type of feature structure. Each feature structure declaration in turn has three parts: an optional description (which gives a prose comment on what that type of feature structure encodes), an obligatory set of feature declarations (which specify range constraints and default values for the features in that type of structure), and optional feature structure constraints (which specify *inter alia* co-occurrence restrictions on feature values). It is recommended that the header be encoded as a

<teiHeader> (see *the TEI Guidelines P5*, Chapter 2, The TEI Header[14]). The other components listed above are unique to feature system declarations. Thus, the following new elements are involved:

— **<fsd>** (feature system declaration) contains a feature system declaration.

— **<fsDecl>** (feature structure declaration) declares one type of feature structure.

— **<fsDescr>** [feature structure description (in FSD)] describes in prose what is represented by the type of feature structure declared in the enclosing <fsDecl>.

— **<fDecl>** (feature declaration) declares a single feature, specifying its name, organization, range of allowed values, and optionally its default value.

— **<fsConstraints>** (feature-structure constraints) specifies certain other constraints on valid feature structures within this FSD.

Feature declarations and feature structure constraints are described in the next two subclauses (8.4 and 8.5). The specification of similar <fsDecl> elements can be simplified by devising an inheritance hierarchy for the feature structure types. Each <fsDecl> may name one or more *baseTypes* from which it inherits feature declarations and constraints (these are often called “supertypes”).

For instance, suppose that <fsDecl type="Basic"> contains <fDecl name="one"> and <fDecl name="two">, and that <fsDecl type="Derived" baseTypes="Basic"> contains just <fDecl name="three">. Then any instance of <fs type="Derived"> must include all three features. This is because <fsDecl type="Derived"> inherits the two feature declarations from <fsDecl type="Basic"> when it specifies a base type of Basic.

EXAMPLE The following example shows the overall structure of a complete FSD:

```
<fsd>
 <teiHeader>
<!-- The header is as for a TEI document -->
 </teiHeader>
 <fsDecl type="SomeName">
  <fsDescr>Describes what this type of fs represents</fsDescr>
  <fDecl name="featureOne">
<!-- The declaration for featureOne -->
  </fDecl>
  <fDecl name="featureTwo">
<!-- The declaration for featureTwo -->
  </fDecl>
  <fsConstraints>
<!-- The feature structure constraints go here -->
  </fsConstraints>
 </fsDecl>
 <fsDecl type="AnotherType">
<!-- Declare another type of feature structure -->
 </fsDecl>
</fsd>
```

The formal definition of the <fsd> and its components is as follows:

```
element fsd { att.global.attributes, fsd.content }

fsd.content = teiHeader?, fsDecl+

fsDecl = element fsDecl
{
   att.global.attributes
   fsDecl.attributes,
   fsDecl.cont
}
```

```
fsDecl.content = fsDescr?, fDecl+, fsConstraints?

fsDecl.attributes =
    attribute type { data.enumerated },
    attribute baseTypes { list { data.name+ } }?

fsDescr = element fsDescr
{
  att.global.attributes
  fsDescr.content
}

fsDescr.content = macro.limitedContent
```

baseTypes gives the name of one or more types from which this type inherits feature specifications and constraints; if this type includes a feature specification with the same name as one inherited from any of the types specified by this attribute, or if more than one specification of the same name is inherited, then the possible values of that feature are determined by unification. Similarly, the set of constraints applicable is derived by conjoining those specified explicitly within this element with those implied by the *baseTypes* attribute. When no base type is specified, no feature specification or constraint is inherited.

Although this part of ISO 24610 does provide for default feature values, feature inheritance is defined to be monotonic.

The process of combining constraints may result in a contradiction, for example if two specifications for the same feature specify disjoint ranges of values, and at least one such specification is mandatory. In such a case, there is no valid feature structure of the type being defined.

Every type specified in *baseTypes* must be a single word which is a legal XML name; for example, they cannot include whitespace or begin with digits. Multiple base types are separated with spaces, e.g. <fsDecl type="Sub" baseTypes="Super1 Super2">.

<fsDescr> may contain any prose except certain elements used for transcribing extant texts, e.g. *del*.

8.4 Feature declarations

8.4.1 General

Each feature is declared in an <fDecl> element whose name attribute identifies the feature being declared; this matches the name attribute of the <f> elements it declares.

An <fDecl> has three parts: an optional prose description, which should explain what the feature and its values represent, an obligatory range specification, which declares what values the feature is allowed to have, and an optional default specification, which declares what default value should be supplied when the named feature does not appear in an <fs>. A single unconditional default value or multiple conditional values may be specified.

8.4.2 Type inference for obligatory features

If, in a feature structure, a feature

— is not optional (i.e. is obligatory),

— has no value provided, or the value <default> is provided (see ISO 24610-1:2006, 5.10, *Default values*), and

— either has no default specified, or has conditional defaults, none of the conditions on which is met,

then the value of this feature in the feature structure's most general valid extension is the most general value provided in its <vRange> in the case of a unit organization, or the singleton set, bag or list containing that element in the case of a complex organization.

8.4.3 Type inference for optional features with defaults

If, in a feature structure, a feature

— is optional,

— has no value provided, or the value <default> is provided, and

— either has a default specified, or has conditional defaults, one of the conditions on which is met,

then this feature does have a value in the feature structure's most general valid extension when it exists, namely the default value that pertains. Naturally, the feature also takes on this value if it is obligatory and a default is specified.

8.4.4 Type inference for optional features without defaults

If, in a feature structure, a feature

— is optional,

— has no value provided, or the value <default> is provided, and

— has no default specified, or has conditional defaults, none of the conditions on which is met,

then this feature does not have a value in the feature structure's most general valid extension, when it exists. This is permitted, because it is optional.

8.4.5 Possibility of failed inference

It is possible that a feature structure will not have a valid extension because the default value that pertains to a feature is not consistent with that feature's declared range. Additional tools are required for the enforcement of such criteria.

8.4.6 Elements and attributes of feature declarations

<fDecl> (feature declaration) declares a single feature, specifying its name, organization, range of allowed values, an optional default value, and whether or not the feature itself is optional. The following elements and attributes are used in feature declarations.

— **name** indicates the name of the feature being declared; it matches the name attribute of <f> elements in the text.

— **org** specifies the organizing discipline of the feature value.

— **optional** indicates whether or not the feature is optional in feature structures of the type being declared.

— **<fDescr>** [feature description (in FSD)] describes in prose what is represented by the feature being declared and its values.

— **<vRange>** (value range) defines the range of allowed values for a feature, in the form of an <fs>, <vAlt>, or built-in; for the value of an <f> to be valid, it must be subsumed by the specified range; if the <f> contains multiple values (as sanctioned by the *org* attribute), then each value must be subsumed by the vRange.

— **<vDefault>** (value default) declares the default value to be supplied when a feature structure does not contain an instance of <f> for this name; if unconditional, it is specified as one (or, depending on the value of the *org* attribute of the enclosing fDecl) more <fs> elements or primitive values; if conditional, it is specified as one or more if elements; if no default is specified, or no condition matches, the value none is assumed.

— **<if>** defines a conditional default value for a feature; the condition is specified as a feature structure, and is met if it subsumes the feature structure in the text for which a default value is sought.

— **<then>** separates the condition from the default in an <if>, or the antecedent and the consequent in a <cond> element.

8.4.7 Feature declarations and subsumption

The logic for validating feature values and for matching the conditions for supplying default values is based upon the operation of subsumption. Subsumption is a standard operation in feature-structure-based formalisms. Informally, a feature structure FS subsumes all feature structures that are consistent with and at least as informative as itself; that is, all feature structures that specify all of the feature values that FS does with values that are subsumed by the values that FS has, and that have all of the re-entrancies that FS does (Carpenter, 1992[1]). See 5.1 for a formal definition.

Following the spirit of the informal definition above, we can extend subsumption in a straightforward way to cover alternation, negation, special primitive values, and the use of attributes in the markup – for instance, a <vAlt> containing the value v subsumes v. The negation of a value v (represented by means of the <vNot> element discussed in ISO 24610-1:2006, 5.9.3 *Negation*) subsumes any value that does not unify with v or, in the case of alternations and negations, does not include v; for example

```
<vNot>
 <numeric value="0"/>
</vNot>
```

subsumes any numeric value other than zero.

The value, <**fs type**="X"/>, even if it is not valid, subsumes any feature structure of type X.

8.4.8 Example of feature declarations

8.4.8.1 As an example of feature declarations, consider the following extract from Gazdar *et al*.s Generalized Phrase Structure Grammar (GPSG)[4]. In the appendix to their book (pp. 245-247), they propose a feature system for English of which this is just a sampling:

Feature value ranges:

— INV {+, -}

— SUBJ {+, -}

— CONJ {and, both, but, either, neither, nor, or, NIL}

— COMP {for, that, whether, if, NIL}

— AGR CAT

— PFORM {to, by, for, . . . }

Feature specification defaults:

— FSD 1: [-INV]

— FSD 2: ⌣[CONJ]

— FSD 9: [INF, +SUBJ] → [COMP for]

8.4.8.2 Note that “FSD” here does not refer to the feature system declarations of this International Standard, but to GPSG's feature specification defaults. The INV feature, which encodes whether or not the subject-verb order in a sentence is inverted, allows only the values *plus* (+) and *minus* (–). If the feature is not specified, then the default rule (FSD 1 above) says that a value of *minus* is always assumed. The feature declaration for this feature would be encoded as follows:

```
<fDecl name="inv">
 <fDescr>inverted sentence</fDescr>
 <vRange>
  <vAlt>
   <binary value="true"/>
   <binary value="false"/>
  </vAlt>
 </vRange>
 <vDefault>
  <binary value="false"/>
 </vDefault>
</fDecl>
```

The value range is specified as an alternation (more precisely, an exclusive disjunction) between values that can be represented by the <binary> feature value. That is, the value must be either true or false, but cannot be both or neither.

8.4.8.3 The CONJ feature indicates the surface form of the conjunction used in a construction. The ~ in the default rule (see FSD 2 above) represents negation. This means that by default the feature is not present, in other words, no conjunction is taking place.

Note that CONJ not being present is different from CONJ being present and having the NIL value allowed in the value range. In their analysis, NIL means that the phenomenon of conjunction is taking place but there is no explicit conjunction in the surface form of the sentence. The feature declaration for this feature would be encoded as follows:

```
<fDecl name="conj">
 <fDescr>surface form of the conjunction</fDescr>
 <vRange>
  <vAlt>
   <symbol value="and"/>
   <symbol value="both"/>
   <symbol value="but"/>
   <symbol value="either"/>
   <symbol value="neither"/>
   <symbol value="nor"/>
   <symbol value="or"/>
   <symbol value="NIL"/>
   <binary value="false"/>
  </vAlt>
 </vRange>
 <vDefault>
  <binary value="false"/>
 </vDefault>
</fDecl>
```

Note that the <vDefault> is not strictly necessary in this case, since the binary value of false only serves to convey the information that the feature has no other legitimate value.

8.4.8.4 The COMP feature indicates the surface form of the complementizer used in a construction. In its range of values, it is analogous to CONJ. However, its default rule (see FSD 9 above) is conditional. It says that if the verb form is infinitival (the VFORM feature is not mentioned in the rule since it is the only feature that can take INF as a value), and the construction has a subject, then a *for* complement must be used. For

instance, to make John the subject of the infinitive in “It is necessary to go”, a *for* complement must be used; that is, It is necessary *for* John to go. The feature declaration for this feature would be encoded as follows:

```
<fDecl name="comp">
 <fDescr>surface form of the complementizer</fDescr>
 <vRange>
  <vAlt>
   <symbol value="for"/>
   <symbol value="that"/>
   <symbol value="whether"/>
   <symbol value="if"/>
   <symbol value="NIL"/>
  </vAlt>
 </vRange>
 <vDefault>
  <if>
   <fs>
    <f name="vform">
     <symbol value="INF"/>
    </f>
    <f name="subj">
     <binary value="true"/>
    </f>
   </fs>
   <then/>
   <symbol value="for"/>
  </if>
 </vDefault>
</fDecl>
```

8.4.8.5 The AGR feature stores the features relevant to subject-verb agreement. Gazdar *et al*. (1985) specify the range of this feature as CAT. This means that the value is a category, which is their term for a feature structure. This is actually too weak a statement. Not just any feature structure is allowable here; it must be a feature structure for agreement (which is defined, in the complete example at the end of that chapter of Gazdar *et al*. (1985)[4], to contain the features of person and number). The following feature declaration encodes this constraint on the value range:

```
<fDecl name="agr">
 <fDescr>agreement for person and number</fDescr>
 <vRange>
  <fs type="Agreement"/>
 </vRange>
</fDecl>
```

That is, the value must be a feature structure of type *Agreement*. The complete example in Annex A provides the <fsDecl type="Agreement"> which includes <fDecl name="pers"> and <fDecl name="num">.

8.4.8.6 The PFORM feature indicates the surface form of the preposition used in a construction. Since PFORM is specified above as an open set, <string> is used in the range specification below rather than <symbol>.

```
<fDecl name="pform">
 <fDescr>word form of a preposition</fDescr>
 <vRange>
  <vNot>
   <string/>
  </vNot>
 </vRange>
</fDecl>
```

EXAMPLE This example makes use of a negated value:

```
<vNot>
 <string/>
</vNot>
```

subsumes any string that is not the empty string.

8.4.8.7 The formal definition for feature declarations follows. Note that the class model.featureVal includes all possible feature values, including feature structures, alternations (<vAlt>), and complex collections (<vColl>).

```
fDecl = element fDecl
{
  att.global.attributes,
  fDecl.attributes,
  fDecl.content
}

fDecl.attributes =
    attribute name { data.name },
   attribute optional { xsd:boolean }?,
   attribute org { "unit" | "set" | "bag" | "list" }?

fDecl.content = fDescr?, vRange, vDefault?

fDescr = element fDescr
{
  att.global.attributes,
  macro.limitedContent
}

vRange = element vRange
{
  att.global.attributes,
  model.featureVal
}

vDefault = element vDefault
{
  att.global.attributes,
  ( model.featureVal+ | if+ )
}

if = element if
{
  att.global.attributes,
  ( ( fs | f ), then, ( model.featureVal ) )
}

then = element then
{
  att.global.attributes,
  empty
}
```

8.5 Feature structure constraints

Ensuring the validity of feature structures may require much more than simply specifying the range of allowed values for each feature. There may be constraints on the co-occurrence of one feature value with the value of another feature in the same feature structure or in an embedded feature structure.

Such constraints on valid feature structures are expressed as a series of conditional and biconditional tests in the <fsConstraints> part of an <fsDecl>. A particular feature structure is valid only if it meets all the constraints. The <cond> element encodes the conventional if-then conditional of Boolean logic which succeeds when either the consequent is true or the antecedent is false. The <bicond> element encodes the biconditional (if and only if) operation of Boolean logic. It succeeds only when the corresponding if-then conditionals in both directions are true. In feature structure constraints, the antecedent and consequent are expressed as feature structures; they are considered true if their feature structure subsumes (see 8.4, *Feature declarations*) the feature structure in question. Procedurally, if the antecedent is true, then the consequent must also be true, so the truth of the consequent is asserted rather than simply tested. A conditional is thus enforced by determining that the antecedent does not (and will never) subsume the given feature structure, or by determining that the antecedent does subsume the given feature structure, and then unifying the consequent with it (the result of which, if successful, will be subsumed by the consequent). In practice, the enforcement of such constraints can result in periods in which the truth of a constraint with respect to a given feature structure is simply not known; in this case, the constraint must be persistently monitored as the feature structure becomes more informative until either its truth value is determined or computation fails for some other reason.

The following elements make up the <fsConstraints> part of an FSD:

— **fsConstraints** (feature-structure constraints) specifies constraints on the content of valid feature structures.

— **cond** (conditional feature-structure constraint) defines a conditional feature-structure constraint; the consequent and the antecedent are specified as feature structures or feature-structure collections; the constraint is satisfied if both the antecedent and the consequent subsume the given feature structure, or if the antecedent does not.

— **bicond** (biconditional feature-structure constraint) defines a biconditional feature-structure constraint; both consequent and antecedent are specified as feature structures or collocations of feature structures; the constraint is satisfied if both subsume a given feature structure, or if both do not.

— **then** separates the condition from the default in an if, or the antecedent and the consequent in a <cond> element.

— **iff** separates the condition from the consequence in a <bicond> element.

For an example of feature structure constraints, consider the following “feature co-occurrence restrictions” extracted from the feature system for English proposed by Gazdar *et al*. (1985:246):

— FCR 1: [+INV] → [+AUX, FIN]

— FCR 7: [BAR 0] ↔ [N] & [V] & [SUBCAT]

— FCR 8: [BAR 1] → ~[SUBCAT]

The first constraint says that if a construction is inverted, it must also have an auxiliary and a finite verb form. That is:

```
<cond>
 <fs>
  <f name="inv">
   <binary value="true"/>
  </f>
 </fs>
 <then/>
 <fs>
  <f name="aux">
   <binary value="true"/>
  </f>
  <f name="vform">
   <symbol value="fin"/>
  </f>
 </fs>
</cond>
```

The second constraint says that if a construction has a BAR value of zero (i.e. it is a lexical item), then it must have a value for the features N, V, and SUBCAT. By the same token, because it is a biconditional, if it has values for N, V, and SUBCAT, it must have BAR='0'. That is:

```
<bicond>
 <fs>
  <f name="bar">
   <symbol value="0"/>
  </f>
 </fs>
 <iff/>
 <fs>
  <f name="n"/>
  <f name="v"/>
  <f name="subcat"/>
 </fs>
</bicond>
```

NOTE Here, according ISO 24610-1:2006, 5.10, *Default values*, (107), <f name="n"> is, for instance, understood as having a value in the possible range of its values, thus being equivalent to the following:

```
<f name="n">
 <vAlt>
  <binary value="true"/>
  <binary value="false"/>
 </vAlt>
</f>
```

The final constraint says that if a construction has a BAR value of 1 (i.e. it is a phrase), then the SUBCAT feature should be absent (~). This is not biconditional, since there are other instances under which the SUBCAT feature is inappropriate. That is:

```
<cond>
 <fs>
  <f name="bar">
   <symbol value="1"/>
  </f>
 </fs>
 <then/>
 <fs>
  <f name="subcat">
   <binary value="false"/>
  </f>
 </fs>
</cond>
```

The formal declaration for feature structure constraints is as follows. Note that <cond> and <bicond> use the empty tags <then> and <iff>, respectively, to separate the antecedent and consequent. These are primarily for the sake of enhancing human readability.

```
fsConstraints = element fsConstraints
{
   att.global.attributes,
  ( cond | bicond )*
}

cond = element cond
{
  att.global.attributes,
  ( ( fs | f ), then, ( fs | f ) )
}

bicond = element bicond
{
  att.global.attributes,
  ( ( fs | f ), iff, ( fs | f ) )
}

iff = element iff
{
  att.global.attributes,
  empty
}
```

Annex A
(normative)

XML schema for feature structures

```
macro.limitedContent = (text | model.limitedPhrase | model.inter)*
macro.xtext = (text | model.gLike)*

att.global.attributes =
  att.global.attribute.xmlid,
  att.global.attribute.n,
  att.global.attribute.xmllang,
  att.global.attribute.xmlbase,
  empty

att.global.attribute.xmlid =
  ## (identifier) provides a unique identifier for the element bearing the
  ##       attribute.
  attribute xml:id { xsd:ID }?

att.global.attribute.n =
  ## (number) gives a number (or other label) for an element, which is
  ##       not necessarily unique within the document.
  attribute n {
    list {
      xsd:token { pattern = "(\p{L}|\p{N}|\p{P}|\p{S})+" }+
    }
  }?

att.global.attribute.xmllang =
  ## (language) indicates the language of the element content using a
  ##       tag generated according to BCP 47
  attribute xml:lang { xsd:language }?

att.global.attribute.xmlbase =
  ## provides a base URI reference with which applications can
  ##       resolve relative URI references into absolute URI
  ##       references.
  attribute xml:base { xsd:anyURI }?

model.gLike = notAllowed

model.featureVal.complex = fs | vColl | vNot | vMerge

model.featureVal.single =
  binary | symbol | numeric | \string | vLabel | \default | vAlt

model.placeStateLike = notAllowed

model.qLike = notAllowed

model.nameLike = model.placeStateLike

model.featureVal = model.featureVal.complex | model.featureVal.single

model.pPart.data = model.nameLike

model.inter = model.qLike
```

```
model.limitedPhrase = model.pPart.data

fsdDecl =
  ## (feature system declaration) provides a feature system declaration
  ## comprising one or more
  ##   feature structure declarations or feature structure declaration links.
  element fsdDecl {
    (fsDecl | fsdLink)+,
    att.global.attribute.xmlid,
    att.global.attribute.n,
    att.global.attribute.xmllang,
    att.global.attribute.xmlbase,
    empty
  }

fsDecl =
  ## (feature structure declaration) declares one type of feature structure.
  element fsDecl {
    (fsDescr?, fDecl+, fsConstraints?),
    att.global.attribute.xmlid,
    att.global.attribute.n,
    att.global.attribute.xmllang,
    att.global.attribute.xmlbase,

    ## gives a name for the type of feature structure being declared.

    attribute type { xsd:Name },

    ## gives the name of one or more typed feature structures
    ##      from which this type inherits feature specifications and
    ##      constraints; if this type includes a feature specification
    ##      with the same name as that of any of those specified by this
    ##      attribute, or if more than one specification of the same name
    ##      is inherited, then the set of possible values is defined by
    ##      unification. Similarly, the set of constraints applicable is
    ##      derived by combining those specified explicitly within this
    ##      element with those implied by the baseTypes
    ##      attribute. When no baseTypes attribute is specified, no
    ##      feature specification or constraint is inherited.

    attribute baseTypes {
      list { xsd:Name+ }
    }?,
    empty
  }

fsDescr =
  ## (feature system description (in FSD)) describes in prose
  ## what is represented by the type of feature
  ## structure declared in the enclosing fsDecl.
  element fsDescr {
    macro.limitedContent,
    att.global.attribute.xmlid,
    att.global.attribute.n,
    att.global.attribute.xmllang,
    att.global.attribute.xmlbase,
    empty
  }
```

```
fsdLink =
  ## (feature structure declaration link) associates the name of
  ## a typed feature structure with a feature
  ## structure declaration for it.
  element fsdLink {
    empty,
    att.global.attribute.xmlid,
    att.global.attribute.n,
    att.global.attribute.xmllang,
    att.global.attribute.xmlbase,

    ## identifies the type of feature structure to be documented;
    ##       this will be the value of the type attribute on at least one
    ## feature structure.

    attribute type { xsd:Name },

    ## supplies a pointer to a feature structure declaration
    ##       (fsDecl) element within the current document or elsewhere.

    attribute target { xsd:anyURI },
    empty
  }

fDecl =
  ## (feature declaration) declares a single feature,
  ## specifying its name, organization,
  ## range of allowed values, and optionally its default value.
  element fDecl {
    (fDescr?, vRange, vDefault?),
    att.global.attribute.xmlid,
    att.global.attribute.n,
    att.global.attribute.xmllang,
    att.global.attribute.xmlbase,

    ## indicates the name of the feature being declared; matches the
    ## name attribute of f elements in the text.

    attribute name { xsd:Name },

    ## indicates whether or not the value of this feature may
    ##       be present.

    [ a1:defaultValue = "true" ] attribute optional { xsd:boolean }?,
    empty
  }

fDescr =
  ## (feature description (in FSD)) describes in prose
  ## what is represented by the feature being
  ## declared and its values.
  element fDescr {
    macro.limitedContent,
    att.global.attribute.xmlid,
    att.global.attribute.n,
    att.global.attribute.xmllang,
    att.global.attribute.xmlbase,
    empty
  }
```

```
vRange =
  ## (value range) defines the range of allowed values for a feature,
  ## in the form of an fs, vAlt, or primitive value;
  ## for the value of an f to be valid, it must be
  ## subsumed by the specified range; if the f
  ## contains multiple values (as sanctioned by the org attribute),
  ## then each value must be subsumed by the vRange.
  element vRange {
    model.featureVal,
    att.global.attribute.xmlid,
    att.global.attribute.n,
    att.global.attribute.xmllang,
    att.global.attribute.xmlbase,
    empty
  }

vDefault =
  ## (value default) declares the default value to be supplied
  ## when a feature structure
  ## does not contain an instance of f for this name; if
  ## unconditional, it is specified as one (or, depending on the value of
  ## the org attribute of the enclosing fDecl) more
  ## fs elements or primitive values;
  ## if conditional, it is specified as
  ## one or more if elements; if no default is specified, or no
  ## condition matches, the value none is assumed.
  element vDefault {
    (model.featureVal+ | if+),
    att.global.attribute.xmlid,
    att.global.attribute.n,
    att.global.attribute.xmllang,
    att.global.attribute.xmlbase,
    empty
  }

if =
  ## defines a conditional default value for a feature; the condition
  ## is specified as a feature structure, and is met if it
  ## subsumes the feature structure in the text for which a
  ## default value is sought.
  element if {
    ((fs | f), then, model.featureVal),
    att.global.attribute.xmlid,
    att.global.attribute.n,
    att.global.attribute.xmllang,
    att.global.attribute.xmlbase,
    empty
  }

then =
  ## separates the condition from the default in an if, or
  ## the antecedent and the consequent in a cond element.
  element then {
    empty,
    att.global.attribute.xmlid,
    att.global.attribute.n,
    att.global.attribute.xmllang,
    att.global.attribute.xmlbase,
    empty
  }
```

```
fsConstraints =
  ## (feature-structure constraints) specifies constraints
  ## on the content of valid feature
  ## structures.
  element fsConstraints {
    (cond | bicond)*,
    att.global.attribute.xmlid,
    att.global.attribute.n,
    att.global.attribute.xmllang,
    att.global.attribute.xmlbase,
    empty
  }

cond =
  ## (conditional feature-structure constraint) defines a conditional
  ## feature-structure constraint; the consequent
  ## and the antecedent are specified as feature structures or
  ## feature-structure collections; the constraint is satisfied if both the
  ## antecedent and the consequent subsume a given feature
  ## structure, or if the antecedent does not.
  element cond {
    ((fs | f), then, (fs | f)),
    att.global.attribute.xmlid,
    att.global.attribute.n,
    att.global.attribute.xmllang,
    att.global.attribute.xmlbase,
    empty
  }

bicond =
  ## (biconditional feature-structure constraint) defines a
  ## biconditional feature-structure constraint; both
  ## consequent and antecedent are specified as feature structures
  ## or groups of feature structures; the constraint is satisfied
  ## if both subsume a given feature structure, or if both do not.
  element bicond {
    ((fs | f), iff, (fs | f)),
    att.global.attribute.xmlid,
    att.global.attribute.n,
    att.global.attribute.xmllang,
    att.global.attribute.xmlbase,
    empty
  }

iff =
  ## (if and only if) separates the condition from the consequence
  ## in a bicond element.
  element iff {
    empty,
    att.global.attribute.xmlid,
    att.global.attribute.n,
    att.global.attribute.xmllang,
    att.global.attribute.xmlbase,
    empty
  }
```