eCH-0035: Conception de schémas XML

Titre	Conception de schémas XML
Code	eCH-0035
Type	Bonne pratique
Stade	expérimentale
Version	1.00
Statut	Annulé
Date de décision	
Date de publication	
Remplace	--
Langues	Allemand, français
Auteur	Groupe spécialisé XML Erik Wilde, EPF Zurich, net.dret@dret.net
Editeur / Distributeur	Association eCH, Amtshausgasse 18, 3011 Berne T 031 560 00 20, F 031 560 00 25 www.ech.ch/ info@ech.ch

Résumé

Le présent document a été intégré dans le eCH-0018 à partir de la version 2.0.

Table des matières

1 Etat du document

Annulé: Le document a été retiré de eCH. Il ne doit plus être utilisé.

1.1 Terminologie des recommandations

Les directives formulées dans ce document sont classées selon la terminologie du [RFC2119]; les expressions suivantes, écrites en LETTRES CAPITALES, sont utilisées avec les significations indiquées ci-dessous (citation du [RFC2119]):

- **MUST**: This word, or the terms «**REQUIRED**» or «**SHALL**», mean that the definition is an absolute requirement of the specification.
- **MUST NOT**: This phrase, or the phrase «**SHALL NOT**», mean that that definition is an absolute prohibition of the specification.
- **SHOULD**: This word, or the adjective «**RECOMMENDED**», mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
- **SHOULD NOT**: This phrase, or the phrase «**NOT RECOMMENDED**» mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
- **MAY**: This word, or the adjective «**OPTIONAL**», mean that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides.)

2 Résumé des recommandations

2.1 Versions de XML et de XML Schema

Recommandations du paragraphe 4

- **SHOULD**: Comme XML 1.1 n'est nécessaire que dans des cas d'application très particuliers, on devrait utiliser XML 1.0 et XML Schema 1.0 dans la mesure du possible.
- **MAY**: Si l'utilisation de XML 1.1 est impérativement requise, le traitement de XML 1.1 peut être combiné avec XML Schema 1.0 de la manière définie dans la note [xml11schema10] du W3C.

2.2 XML Schema et XML Namespaces

Recommandation du paragraphe 5

- **SHOULD**: Le nom d'espace nominatif d'un schéma (comme indiqué dans l'attribut `targetNamespace`) devrait, comme exigé dans [eCH-0018], faire référence non pas directement au schéma, mais à une description de l'espace nominatif défini, selon [eCH-0033]. Cette description comprendra alors, en plus d'autres informations, des références au schéma garantissant que celui-ci peut être retrouvé.

2.3 Modélisation avec XML Schema

Recommandations du paragraphe 6

- **SHOULD**: Dans le traitement par ordinateur de données typisées (c'est-à-dire de presque toutes les données dans les scénarios B2B), le schéma devrait être défini au moyen d'un schéma XML, qui seul permet de définir les types de données dans une forme proche de l'application.
- **MAY**: Si les données échangées sont, dans leur majorité, des données non typisées (documents orientés texte), l'utilisation de DTD peut aussi être adéquate, mais on tiendra alors compte du fait que les DTD prennent en charge des contraintes nettement moins strictes que XML Schema.

2.4 Déclaration locale/globale d'éléments/de types

Recommandations du paragraphe 7

- **SHOULD**: Le modèle de conception le plus utilisé devrait être la structure en stores vénitiens, avec une définition globale de tous les types et une définition locale ou globale, suivant les besoins, des éléments et des attributs (définition globale uniquement s'ils sont réutilisés à différents endroits). Les types correspondent donc à l'aspect d'un schéma qui est mis spécialement en évidence et qui est bien approprié à la réutilisation grâce à la définition globale générale.

- **SHOULD NOT**: Dans la définition locale d'éléments ou d'attributs, on devrait veiller à ce qu'aucun élément ni attribut de même nom ne soit défini localement parce que cela pourrait perturber les utilisateurs du schéma. Quand il s'agit du même élément, une définition globale devrait être faite, puis référencée. Par contre, on ne devrait pas donner le même nom à des concepts différents.
- **MAY**: Si des éléments doivent réutilisés, ils peuvent aussi être définis globalement (cela est p. ex. indispensable pour les récursions). On tiendra toutefois compte du fait que, dans la structure en stores vénitiens, les types représentent le genre principal de réutilisation, de sorte que chaque cas concerné devrait être considéré séparément.
- **SHOULD**: Si un type doit être réutilisé en relation avec des Identity Constraints (comme décrit au paragraphe 9.5), cette réutilisation devrait être mentionnée clairement dans le type lui-même. La meilleure façon de la documenter est de définir, pour ce type, un élément exemple contenant les Identity Constraints. Dans le type, on peut alors indiquer par un commentaire que, en cas de réutilisation du type, les Identity Constraints doivent être copiés depuis l'élément exemple.

2.5 Traitement basé sur les noms ou traitement basé sur les types

Recommandation du paragraphe 8.1

- **MUST**: Si un schéma XML applique des mécanismes obligeant, lorsque l'on travaille avec des instances, d'utiliser des informations de types (c'est-à-dire les mécanismes décrits au paragraphe 8.2), on le mentionnera clairement dans le schéma ainsi que dans la documentation.

2.6 Mécanismes de substitution de types dans XML Schema

Recommandation du paragraphe 8.2

- **MUST**: L'utilisation de l'un des deux mécanismes de substitution (`xsi:type` ou groupes de remplacement) doit être exclusive. Un mélange des deux styles rendrait le schéma difficile à comprendre et en compliquerait inutilement l'utilisation. Cela n'est pas une restriction, car les deux mécanismes assurent la même fonction de base, à savoir permettre d'utiliser dans un modèle de contenu d'autres types que celui de l'élément qui y est mentionné.

2.7 L'attribut xsi: type

Recommandations du paragraphe 8.2.1

- **SHOULD NOT**: Les mécanismes de la substitution de types `xsi:type` ne devraient jamais être utilisés, sauf pour des raisons importantes. Ils rendent plus difficiles le traitement et la compréhension du schéma ainsi que des instances définies par lui.

- **MUST**: S'il est déclaré explicitement dans le schéma que le mécanisme `xsi:type` peut être utilisé ou même que la conception du schéma exige qu'il le soit (types définis comme `abstract=«true»`), cette utilisation est autorisée, mais doit être documentée de manière adéquate. On veillera ensuite à ce que le traitement de ces instances soit basé sur les types, de manière que l'attribution de type ne s'effectue pas via le type d'un élément du schéma, mais via le `xsi:type` de l'instance.
- **MUST**: Si l'on utilise des mécanismes de substitution de types, on le documentera de manière claire et adéquate, de sorte que les utilisateurs du schéma soient bien informés de cet aspect de ce dernier.

2.8 Substitution de types

Recommandations du paragraphe □

- **SHOULD NOT**: Sauf raisons contraires importantes, on ne devrait jamais utiliser les mécanismes des groupes de substitution, car ils rendent plus difficiles le traitement et la compréhension du schéma et des instances définies par celui-ci.
- **SHOULD**: S'il est déclaré explicitement dans le schéma que des groupes de substitution peuvent être utilisés ou même que la conception de celui-ci exige qu'ils le soient (des éléments qui sont des Substitution Group Heads sont définis comme `abstract=«true»`), cette utilisation est autorisée, mais doit être documentée de manière adéquate. On veillera ensuite à ce que le traitement de ces instances tienne systématiquement compte des groupes de substitution.
- **MUST**: Si l'on utilise les mécanismes des groupes de substitution, on le documentera de manière claire et adéquate, de sorte que les utilisateurs du schéma soient bien informés de cet aspect de ce dernier.

2.9 Eléments racines

Recommandations du paragraphe 9.1

- **MAY**: Les schémas peuvent être conçus de manière que plusieurs éléments interviennent comme éléments racines. Ainsi, un schéma peut servir à valider des instances avec différents éléments racines.
- **SHOULD**: Les éléments racines d'un schéma devraient être identifiés comme tels. Etant donné que XML Schema ne définit aucun mécanisme à cet effet, cette identification doit se faire dans un commentaire désignant les éléments racines potentiels.

2.10 Unités de traitement

Recommandation du paragraphe 9.2

- **SHOULD**: Les unités de traitement qui sont choisies lors de la définition d'un schéma devraient se baser sur la manière dont les données sont structurées du point de vue

technique. Leur choix devrait avoir pour objectif de désigner, par balisage XML, les structures importantes pour le traitement, de sorte que les applications n'aient si possible besoin d'aucune méthode extérieure aux technologies XML pour identifier les structures importantes sur le plan technique et pour travailler avec elles.

2.11 Eléments ou attributs?

Recommandations du paragraphe □

- **SHOULD**: Pour la structuration, on devrait toujours utiliser des éléments, car ceux-ci peuvent être complétés de manière simple en cas de besoin, par exemple par des attributs supplémentaires ou par leur subdivision en structures plus fines.
- **MAY**: Les attributs devraient être utilisés avec précaution. Ils sont soumis à des restrictions importantes (pas de répétition, pas de possibilité de structuration plus détaillée) et constituent par conséquent, en cas de modifications ultérieures d'un schéma, un endroit auquel, le cas échéant, les modifications souhaitées ne peuvent plus être exécutées de manière simple.

2.12 Représentation de valeurs vides

Recommandations du paragraphe 9.4

- **SHOULD**: Les valeurs nulles devraient être représentées au moyen d'éléments ou d'attributs optionnels, c'est-à-dire par l'absence des composantes concernées ou par un contenu vide pour celles-ci.
- **SHOULD NOT**: Le mécanisme `xsi:null` ne devrait pas être utilisé.

2.13 Identification et références

Recommandations du paragraphe 9.5

- **SHOULD**: Si l'univocité, l'existence ou le référencement de noms doivent être garantis, on devrait utiliser des Identity Constraints de XML Schema pour définir les contraintes correspondantes. On veillera à définir alors les XPaths de manière aussi restrictive que possible, afin d'éviter tout conflit en cas de modification ultérieure du schéma.
- **SHOULD**: Si des Identity Constraints sont utilisées, les structures auxquelles elles renvoient devraient utiliser leurs propres types pour les noms. De cette manière, les types peuvent être réutilisés de manière simple si les noms correspondants doivent être repris à un autre endroit du schéma.
- **SHOULD NOT**: Le mécanisme DTD des attributs `ID/IDREF` ne devrait pas être utilisé, car il est soumis à de fortes restrictions et ne permet en outre pas de définir les contraintes de manière aussi exacte que cela est possible avec les Identity Constraints.

- **MAY**: Si l'application fixe des exigences (p. ex. univocité ou référencement sur plusieurs documents) qui ne peuvent pas être implémentées par des Identity Constraints de XML Schema, celles-ci peuvent être définies par une identification et des contraintes externes. On utilisera alors des types spécifiques pour les structures touchées par ces définitions externes et on documentera clairement dans le schéma que les valeurs de ces types sont soumises à des contraintes définies en dehors du schéma.
- **SHOULD**: Si des Identity Constraints doivent intervenir dans la réutilisation de types déterminés, on le documentera de préférence en définissant, pour le type concerné, un élément exemple contenant les Identity Constraints. On pourra alors ajouter un commentaire dans le type pour signaler qu'en cas de réutilisation de celui-ci l'on devra copier les Identity Constraints de l'élément exemple.

2.14 Listes de valeurs

Recommandations du paragraphe 9.6

- **SHOULD**: Si les listes de valeurs sont des listes statiques dont les valeurs sont connues de manière exhaustive et resteront stables en principe, celles-ci devraient être énumérées dans le schéma au moyen d'un Simple Type et d'Enumeration Facets. Pour mieux gérer ces listes des valeurs et pouvoir procéder à de meilleurs contrôles d'accès, il est en général judicieux de les externaliser dans un document de schéma autonome (voir paragraphe 11.1).
- **SHOULD**: Si les listes des valeurs sont des listes dynamiques dont les valeurs ne sont pas connues de manière exhaustive ou peuvent être modifiées, elles ne devraient être définies que par une restriction lexicale la plus exacte possible et par la référence à une liste externe. De nouvelles valeurs peuvent ainsi être ajoutées aux listes des valeurs sans qu'il faille modifier quelque chose au schéma.

2.15 Marquage de la langue de contenus

Recommandation du paragraphe 9.7

- **SHOULD**: Si des marquages de la langue sont nécessaires pour des contenus, ils devraient être réalisés dans un attribut `xml:lang`, car celui-ci est défini dans la norme XML elle-même et constitue une convention générale pour les marquages de langue. [eCH-0050] définit un schéma correspondant, qui est mis à disposition pour réutilisation dans le cadre de eCH.

2.16 Attribution des versions aux schémas XML

Recommandations du paragraphe 10

- **SHOULD**: Une nouvelle version mineure devrait être générée si les instances suivant l'ancien schéma peuvent être traitées correctement lors de la validation et de l'interprétation selon le schéma modifié.
- **SHOULD**: Une nouvelle version majeure devrait être générée si les instances suivant l'ancien schéma ne peuvent plus être traitées correctement lors de la validation et de l'interprétation selon le schéma modifié.
- **SHOULD NOT**: Si la nouvelle version interdit des valeurs ou structures permises auparavant, cette interdiction ne devrait pas être réalisée dans la définition du schéma, mais être signalée comme *deprecated* au niveau de l'application, de manière que les anciennes instances puissent continuer d'être traitées.
- **MUST**: Pour permettre l'identification de la version mineure d'un schéma et de ses instances (la version majeure étant identifiée par le nom d'espace nominatif), on utilisera l'attribut `version` du schéma qui contient uniquement la version mineure. Dans les instances, la version mineure fait l'objet d'un marquage qui doit être prévu dans le schéma lui-même, mais qui devrait de préférence être réalisé par l'utilisation de l'attribut prévu dans [eCH-0050] (cet attribut est aussi recommandé dans [eCH-0018]).
- **MUST**: La version majeure prise en charge des deux côtés doit concorder en cas d'échange de données si celui-ci est bidirectionnel. Si tel n'est pas le cas, la communication ne peut pas avoir lieu.
- **MAY**: S'il s'agit d'un échange de données essentiellement unidirectionnel, on peut éventuellement en tenir compte lors de la définition de la version et, si une version mineure est utilisée, des modifications peuvent être acceptées tant qu'elles n'entraînent aucun problème de compatibilité du côté du destinataire des données. Il est alors nécessaire de réaliser une bonne documentation, décrivant clairement le scénario et définissant ainsi les cas dans lesquels une communication est possible.
- **MAY**: Les recommandations ci-dessus ne sont applicables que pour les schémas développés et utilisés de manière productive. Une procédure simplifiée est possible pour la phase de développement avant l'utilisation productive et pour les schémas non destinés à une telle utilisation.

2.17 Structuration de documents de schéma

Recommandations du paragraphe 11.1

- **SHOULD**: Pour des raisons de modularisation, les documents de schéma dépassant une certaine taille devraient être structurés par l'utilisation de `xs:include`. Cela permet de réaliser des schémas plus clairs ainsi que d'en réutiliser des parties (p. ex. des listes de valeurs, comme décrit au paragraphe 9.6).
- **SHOULD NOT**: Le mécanisme `xs:redefine` de XML Schema ne devrait pas être utilisé, parce qu'il cause des dépendances compliquées entre différents documents de schéma et, surtout, qu'il peut provoquer, en cas de modification ultérieure de documents de schéma, des erreurs difficiles à découvrir.

- **SHOULD NOT**: Les schémas caméléons (c'est-à-dire les documents de schéma sans `targetNamespace`, qui prennent par conséquent l'espace nominatif du document de schéma dans lequel ils sont intégrés) ne devraient pas être utilisés, car ils comportent diverses possibilités d'erreurs et ne signalent pas les points qu'ils ont en commun.
- **MAY**: La modularisation d'un schéma peut aussi être utile si plusieurs groupes d'une équipe travaillent au développement ou à l'extension de celui-ci. Elle sera alors conçue en fonction de l'attribution, au niveau du contenu, des différentes parties du schéma aux différents groupes de l'équipe.

2.18 Structuration de schémas

Recommandations du paragraphe 11.2

- **SHOULD**: Dans la mesure du possible, des solutions déjà disponibles pour des aspects partiels d'un nouveau schéma devraient pouvoir y être réutilisées par importation des schémas concernés. Cette possibilité facilite le développement de schémas et constitue aussi une aide précieuse pour les utilisateurs de schémas, car ils se servent de schémas déjà connus et peuvent ainsi réutiliser du savoir-faire et du code déjà existants.
- **SHOULD**: Si des équipes ou des projets différents participent au développement, ils ne devraient pas réaliser un schéma commun, mais plusieurs schémas distincts. Cette manière de faire permet de mieux découpler le développement et d'éviter les problèmes que pourrait causer une progression plus ou moins rapide dans les équipes ou projets concernés.

2.19 Implémentation de relations référentielles

Recommandations du paragraphe 12

- **MAY**: S'il est indiqué, pour des raisons concernant l'implémentation, de renoncer aux représentations hiérarchiques (parce que par exemple la production et la consommation des données passent toujours par un gestionnaire de bases de données relationnelles), il est autorisé de renoncer à la représentation hiérarchique des données. On ne devrait toutefois le faire qu'avec attention et précaution, parce que les contraintes peuvent être modifiées pour des raisons d'implémentation (p. ex. si l'on passe à des bases de données XML) et que le schéma ne sera alors plus adapté au nouvel environnement.
- **SHOULD**: Si des références sont utilisées, les contraintes auxquelles elles sont soumises devraient être décrites aussi bien que possible à l'aide d'Identity Constraints (avec la réserve faite au paragraphe 9.5) et les contraintes supplémentaires devraient être documentées.
- **SHOULD**: Si des schémas sont conçus essentiellement pour l'utilisateur final, p. ex. dans le cas de documents XML qui sont utilisés (étudiés ou édités) par des per-

sonnes, tels que des fichiers de configuration ou des documents orientés Web, la modélisation aux endroits concernés devrait plutôt être hiérarchique. Les structures hiérarchiques sont plus simples à comprendre pour les personnes que celles qui sont interconnectées entre elles par de nombreuses références.

- **SHOULD**: Si des schémas sont conçus essentiellement pour le traitement par ordinateur, le modèle peut être défini de manière plus plate que pour le cas essentiellement destiné à l'utilisateur final. Aux endroits où existe une structure hiérarchique dûment justifiée, inhérente au modèle de base, une telle structure devrait toutefois aussi être définie dans le schéma XML.

2.20 Schémas XML ouverts

Recommandations du paragraphe 13.1

- **MAY**: Si un schéma ouvert est souhaité, de sorte que peuvent apparaître, dans des instances de celui-ci, des contenus qui n'y sont pas définis en détail, cela peut être réalisé au moyen de jokers (wildcards) pour les éléments et/ou les attributs. Ces jokers peuvent servir d'une part à conserver l'ouverture par rapport aux contenus non connus, mais d'autre part aussi à atteindre l'ouverture envers les futures extensions prévues.

3 Introduction

Le document eCH «Meilleures pratiques XML» [eCH-0018] définit des directives pour l'utilisation de documents XML en général, de documents XML en tant qu'instances de schémas XML existants ainsi que pour la dénomination de composantes dans les schémas XML. Il formule ainsi les principales directives prescrivant un cadre à l'utilisateur de schémas XML existants.

Le présent document va au-delà de ces directives et définit les recommandations qui devraient être respectées lors de la conception de schémas XML. Ces recommandations visent à définir des schémas XML qui puissent être compris le plus facilement possible par l'utilisateur et qui permettent et encouragent la réutilisation de modules de schémas.

3.1 Vue d'ensemble

XML Schema a non seulement apporté une meilleure performance qu'avec les DTD (Document Type Definitions), mais a surtout fortement accru la complexité du langage des schémas. Cela est dû au fait que les éléments du langage sont plus nombreux, offrant ainsi des possibilités beaucoup plus vastes pour l'implémentation des définitions de schémas.

Cela a pour avantage que le concepteur peut comparer les contraintes auxquelles sont soumises les différentes possibilités d'implémentation pour choisir la variante correspondant à ses souhaits. D'autre part, de par la grande complexité de XML Schema, seuls peu d'utilisateurs sont à même de reconnaître toutes les contraintes à respecter et d'en évaluer les effets et interactions.

Par conséquent, XML Schema n'est souvent utilisé que de manière hésitante ou non optimale, parce que l'étude et l'utilisation de ce langage nécessitent un effort important. Pour alléger cet effort, le présent document formule des directives aidant, dans de nombreux cas, à choisir la variante de modélisation qui convient parmi toutes celles qui sont possibles.

3.2 Champ d'application

Le champ d'application de ce document est la conception de schémas XML, donc non pas les questions concernant la manière de réaliser le balisage (qui sont discutées dans le document eCH «Meilleures pratiques XML» [eCH-0018]), mais la manière de réaliser la structure interne du schéma XML ainsi que la structure de ses composantes et des relations qu'elles ont entre elles.

Cela est important notamment s'il s'agit non seulement de traiter des instances du schéma, mais aussi de réutiliser le schéma lui-même, p. ex. pour en reprendre des parties dans un nouveau contexte ou en définir une nouvelle version. Dans les deux cas, la structure interne du schéma influe fortement sur la simplicité d'exécution du travail.

3.3 Avantages

Si des schémas sont définis dès le début pour être réutilisés, cela encourage leur réutilisation et contribue à éviter le développement de solutions parallèles. Lorsque l'on définit des schémas, on devrait toujours le faire dans la perspective qu'ils pourraient être utiles comme éléments de solution dans un cas d'application non prévu, et le schéma devrait être conçu pour ce cas.

3.4 Objets principaux

Le présent document a pour principaux objets de définir des directives et de donner des indications pour la conception de schémas XML. La question de la «bonne» ou de la «mauvaise» conception d'un schéma XML n'a été discutée jusqu'ici que de manière incomplète, même parmi les spécialistes. C'est pourquoi le présent document ne peut pas définir de règles inattaquables.

Nous essayerons en revanche, après avoir défini chaque directive, d'expliquer pourquoi nous l'avons fait. Ainsi, si notre argumentation n'est pas pertinente dans un cas d'application particulier, une autre décision peut alors parfaitement être prise.

4 Versions de XML et de XML Schema

XML existe actuellement en deux versions (1.0 et 1.1, comme décrit au paragraphe 4.1), et comme XML Schema lui est étroitement lié, la question se pose de savoir comment les versions de XML influent sur l'utilisation de XML Schema (qui n'existe jusqu'ici qu'en une seule version).

4.1 Versions de XML

XML a été normalisé dans la première version 1.0 de 1998. Nous en sommes maintenant à la «Third Edition» [xml10third], qui est parue en 2004, mais les différentes «éditions» de cette norme ne comportent aucune différence fonctionnelle. Les «éditions» des normes du W3C correspondent simplement à des versions rédactionnelles de la même norme, dans lesquelles des erreurs et des formulations peu claires ou prêtant à confusion ont été éliminées.

Toutefois, une nouvelle version 1.1 de la norme XML [xml11] a aussi été publiée en 2004, surtout pour harmoniser XML avec une nouvelle version Unicode. En outre, les prescriptions régissant la définition des noms XML ont été assouplies et un caractère supplémentaire a été autorisé comme fin de ligne. Bien qu'elle ne soient pas très grandes, les modifications apportées à la version 1.1 de XML influencent la définition de documents «bien formés», c'est pourquoi une nouvelle version de XML a dû être définie. Sur la base des règles qui lui sont fixées, un processeur XML 1.0 va rejeter tout document XML 1.1, ne serait-ce qu'à cause de l'indication `version=«1.1»` se trouvant éventuellement dans la déclaration XML.

Etant donné que la norme XML Namespaces [xmlns] est en relation étroite avec XML et est directement liée à la version XML, il a aussi fallu en créer une nouvelle version [xmlns11] pour XML 1.1. Cette nouvelle version n'apporte qu'une seule nouveauté fonctionnelle, à savoir la possibilité d'annuler la déclaration d'espaces nominatifs (ce qui n'est autorisé que pour l'espace nominatif par défaut dans XML Namespaces 1.0). Cela mis à part, elle ne constitue qu'une adaptation de la norme XML Namespaces à la norme XML 1.1.

4.2 Versions de XML Schema

XML Schema a été publié pour la première fois en 2001, une «Second Edition» [xmlschema1sec, xmlschema2sec] ayant suivi en 2004 après la découverte de quelques erreurs et questions non réglées dans la première version (surtout en relation avec le développement de XQuery, qui repose fortement sur XML Schema). Comme nous l'avons déjà mentionné en parlant des versions XML, il ne s'agit ici donc pas d'une nouvelle version de XML Schema, mais simplement d'une nouvelle rédaction améliorée de la norme initiale. C'est pourquoi XML Schema existe en une seule version, qui se réfère explicitement à XML 1.0 et à XML Namespaces 1.0, c'est-à-dire qu'un processeur XML Schema actuel et suivant strictement les règles ne peut pas valider de documents XML 1.1.

4.3 Problématique des différentes versions

Etant donné qu'il existe deux versions de XML (paragraphe 4.1), mais une seule de XML Schema (paragraphe 4.2), le traitement de documents XML 1.1 n'est actuellement pas possible avec XML Schema. Un document informel du W3C [xml11schema10] décrit toutefois comment les implémentations existantes de processeurs XML Schema peuvent être modifiées afin qu'elles puissent être utilisées pour la validation de documents XML 1.1. Les modifications décrites sont minimes (adaptation de l'analyse syntaxique XML servant de «front-end» et adaptation des définitions de noms spécifiques à XML 1.0 ainsi que de quelques autres constructions syntaxiques de XML), si bien que le travail nécessaire pour modifier un processeur XML Schema reste relativement modéré.

C'est pourquoi la combinaison de documents XML 1.1 et de XML Schema ne rencontre pas de grands obstacles du point de vue technique; on observera toutefois que l'utilisation de XML 1.1 représente un problème, notamment pour des raisons d'interopérabilité et que la norme [eCH-0018] recommande donc d'utiliser XML 1.1 exclusivement quand il n'est pas possible de trouver une solution basée sur XML 1.0.

4.4 Recommandations

- **SHOULD**: Comme XML 1.1 n'est nécessaire que dans des cas d'application très particuliers, on devrait utiliser XML 1.0 et XML Schema 1.0 dans la mesure du possible.
- **MAY**: Si l'utilisation de XML 1.1 est impérativement requise, le traitement de XML 1.1 peut être combiné avec XML Schema 1.0 de la manière définie dans la note [xml11schema10] du W3C.

5 XML Schema et XML Namespaces

XML Schema est un langage servant à la définition de vocabulaires pour des documents XML. XML Namespaces [xmlns] définit un mécanisme servant à identifier par un nom univoque, à savoir le nom d'espace nominatif, les vocabulaires utilisés dans des documents XML. Le nom d'espace nominatif est une URI [RFC3986] et devrait, dans le contexte de eCH, renvoyer à une description d'espace nominatif selon [eCH-0033] (dans laquelle référence est faite au schéma).

Par l'attribut `targetNamespace`, XML Schema permet d'indiquer déjà dans le schéma lui-même l'espace nominatif du vocabulaire défini. XML Schema exige même que cela soit fait, c'est-à-dire que s'il est utilisé dans des instances avec cet espace nominatif, le vocabulaire doit aussi être défini dans le schéma avec ce `targetNamespace`, sans quoi l'instance ne sera pas validée par rapport au schéma.

L'exemple ci-après (XML Schema pour l'attribut eCH `minorVersion`) illustre l'utilisation de l'attribut `targetNamespace`:

```
<xs:schema xmlns:xs=«http://www.w3.org/2001/XMLSchema» version=«0»
           targetNamespace=«http://www.ech.ch/xmlns/minorversion/v1»>
  <xs:attribute name=«minorVersion» type=«xs:token»/>
</xs:schema>
```

Ce schéma définit un attribut qui l'est comme appartenant à l'espace nominatif XML `http://www.ech.ch/xmlns/minorversion/v1`. Dans chaque instance utilisant cet attribut, l'espace nominatif correspondant doit donc aussi être déclaré et utilisé pour l'attribut `minorVersion`, sans quoi l'attribut ne peut pas être validé par rapport au schéma présenté ici.

5.1 Recommandations

- **SHOULD**: Le nom d'espace nominatif d'un schéma (comme indiqué dans l'attribut `targetNamespace`) devrait, comme exigé dans [eCH-0018], faire référence non pas directement au schéma, mais à une description de l'espace nominatif défini, selon [eCH-0033]. Cette description comprend, en plus d'autres informations, des références au schéma garantissant que celui-ci peut être retrouvé.

6 Modélisation avec XML Schema

XML Schema a été développé pour remplacer les DTD intégrées dans XML. Les deux principales nouveautés de ce langage sont, d'une part, sa bibliothèque riche en types de données simples (partie 2 de la norme [xmlschema2sec]) et, d'autre part, l'introduction d'une couche de types (partie 1 de la norme [xmlschema1sec]), ce qui crée un niveau de modélisation entièrement nouveau par rapport aux DTD (voir figure 1) et place une couche d'abstraction de types par dessus les éléments et les attributs des DTD. Dans XML Schema, les éléments et les attributs sont toujours des instances de types définissant quels contenus ces éléments ou attributs peuvent avoir.

	DTD		XML Schema	
	Exemple	**Terminologie**	**Terminologie**	**Beispiel**
Schéma	`<!ELEMENT test EMPTY>`	Element Type Declaration	Type Definition (Simple or Complex) Element Declaration	`<xs:complexType name="testType"/>` `<xs:element name="test" type="testType"/>`
XML	`<test/>`	Element	Element	`<test/>`

Figure 1: Niveaux de modélisation des DTD et de XML Schema en comparaison

Le niveau des types de XML Schema a souvent été «simulé» au moyen de Parameter Entities (entités qui ne peuvent être utilisée que dans les DTD et qui sont référencées comme `%name;`), mais cela ne constituait qu'une voie très modeste et exigeant une grande autodiscipline pour atteindre une certaine systématique et la réutilisabilité dans les DTD. Dans XML Schema, le niveau de types est un élément important du langage et permet la réutilisation (plusieurs éléments utilisent le même type), l'extension (un type étant un type existant) et la restriction (un type restreint un type existant).

6.1 Différences au niveau du traitement

Alors qu'un XML DTD est traité le plus souvent simplement sur la base d'un modèle DOM, le traitement d'un XML basé sur XML Schema s'effectue idéalement au moyen d'un Toolset offrant des informations de type, c'est-à-dire indiquant toujours directement le type auquel appartiennent les éléments et les attributs, de sorte que le traitement proprement dit peut avoir lieu sur la base de types définis. On se heurte ici toutefois rapidement aux limites des technologies existantes: bien que, par l'interface `TypeInfo` introduite dans DOM3, on en arrive aux informations de types relatives à un élément ou à un attribut, il n'existe encore aucun module DOM à l'aide duquel l'on pourrait suivre cette information de type dans le modèle XML Schema, p. ex. pour constater sur quel type se base le type trouvé. On doit encore utiliser à cet effet des interfaces propriétaires telles que le *Xerces Schema Component Model API* ou le XSD de IBM. On peut prévoir que de nouveaux développements se feront dans ce domaine et que l'on pourra aussi accéder aux informations de schéma par une interface

normalisée, mais cela n'est pas encore réalisé. En réalité, la situation est telle que l'on ne travaille pas parfaitement sur la base de types définis dans la plus grande partie des applications, mais que l'information de types est considérée comme câblée de manière fixe avec les éléments. Cette manière de voir est particulièrement intéressante et problématique quand il s'agit de concevoir des schémas XML de manière à ce qu'ils durent longtemps et qu'ils puissent être réutilisés (ainsi que le logiciel qui se base sur eux).

6.2 Recommandations

Comme le définit [eCH-0018], on devrait, dans la mesure du possible, donner la préférence au traitement sur la base de XML.

- **SHOULD**: Dans le traitement par ordinateur de données typées (c'est-à-dire de presque toutes les données dans les scénarios B2B), le schéma devrait être défini au moyen d'un schéma XML, qui seul permet de définir les types de données dans une forme proche de l'application.
- **MAY**: Si les données échangées sont, dans leur majorité, des données non typées (documents orientés texte), l'utilisation de DTD peut aussi être adéquate, mais on tiendra alors compte du fait que les DTD prennent en charge des contraintes nettement moins strictes que XML Schema.

7 Déclaration locale/globale d'éléments/de types

Tant les éléments ou attributs que les types peuvent être définis localement ou globalement dans XML Schema. Les définitions locales interviennent à l'intérieur d'autres définitions (éléments/attributs dans des types ou des Named Groups, types dans des éléments ou des attributs), alors que les définitions globales se font au niveau supérieur du schéma (comme enfants de l'élément `xs:schema`), portent un nom et sont référencées à un autre endroit. Voyons maintenant quelles sont les différences entre ces concepts:

- *Définitions locales.* Ces définitions interviennent à l'intérieur d'autres définitions, pour les éléments p. ex. dans des types ou des Named Model Groups, pour les types dans des éléments ou des attributs. Les définitions locales ne peuvent pas être réutilisées (car elles ne portent pas de noms référençables), mais améliorent souvent la lisibilité d'un schéma (les utilitaires de conception de schémas relativisent toutefois ce point, car ils intègrent souvent aussi les définitions globales de manière bien reconnaissable dans une représentation graphique du schéma).
- *Définitions globales.* Ces définitions interviennent toujours au niveau supérieur du schéma et sont donc des constructions autonomes (c'est-à-dire non intégrées dans d'autres définitions) qui peuvent être référencées par leurs noms.

En plus de cette question de référenciabilité, il est important de comprendre que l'aspect «local» ou «global» a un effet sur l'attribution des noms (dans les documents!), notamment en ce qui concerne les éléments et les attributs. Alors que les noms globaux doivent toujours apparaître de manière entièrement qualifiée dans un document (c'est-à-dire avec le *Namespace Prefix* si le schéma déclare un *Target Namespace*), les noms locaux sont régis par les attributs `elementFormDefault` et `attributeFormDefault` du schéma (qui exigent par défaut des noms non qualifiés). Cela n'est qu'une remarque préliminaire, nous entrerons plus en détail au chapitre suivant sur la question des espaces nominatifs et de leur utilisation dans XML Schema.

Partant des éléments et types locaux et globaux, différentes variantes de conception peuvent être définies. Nous allons les décrire et les commenter sur la base du petit document d'exemple ci-après.

```
<person>
  <name>
    <givenname>Erik</givenname>
    <givenname>Thomas</givenname>
    <surname>Wilde</surname>
  </name>
  <address>
    <company>ETH Zürich</company>
    <email>net.dret@dret.net</email>
  </address>
</person>
```

7.1 Poupée russe (Russian Doll)

Le schéma en poupées russes réalise une imbrication maximale. Il n'y existe donc qu'une seule définition globale, dans laquelle sont compris tous les autres éléments et types en tant que définitions locales. Son avantage est que la structure des instances y est très reconnaissable. Ses inconvénients sont un haut degré d'imbrication et l'impossibilité de principe de représenter des structures récursives (c'est-à-dire imbriquées en elles-mêmes).

```
<xs:schema xmlns:xs=«http://www.w3.org/2001/XMLSchema»>
 <xs:element name=«person»>
  <xs:complexType>
   <xs:sequence>
    <xs:element name=«name»>
     <xs:complexType>
      <xs:sequence>
       <xs:element name=«givenname» type=«xs:token»
                   maxOccurs=«unbounded»/>
       <xs:element name=«surname» type=«xs:string»/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name=«address» minOccurs=«0»>
     <xs:complexType>
      <xs:sequence>
       <xs:element name=«company» type=«xs:string»/>
       <xs:element name=«email» minOccurs=«0»>
        <xs:simpleType>
          <xs:restriction base=«xs:string»>
           <xs:pattern value=«.*@.*\..*»/>
          </xs:restriction>
        </xs:simpleType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

7.2 Jardin d'Eden (Garden of Eden)

Le schéma en Jardin d'Eden est l'opposé exact de celui en poupées russes. Comme au Jardin d'Eden, tout chose y reçoit un nom et est donc défini globalement. Ainsi, tant les éléments ou attributs que les types sont réutilisables, mais le schéma devient assez volumineux et peu clair (à cause des nombreux attributs `ref` et `type`). Si l'on utilise un utilitaire de conception de schéma, ce manque de clarté est souvent masqué par une interface graphique, mais force est de constater que l'on a ici affaire à un schéma extrême dans sa philosophie.

```
<xs:schema xmlns:xs=«http://www.w3.org/2001/XMLSchema»>
  <xs:element name=«person» type=«personType»/>
  <xs:element name=«name» type=«nameType»/>
  <xs:element name=«givenname» type=«givennameType»/>
```

```
  <xs:element name=«surname» type=«surnameType»/>
  <xs:element name=«address» type=«addressType»/>
  <xs:element name=«company» type=«companyType»/>
  <xs:element name=«email» type=«emailType»/>
  <xs:complexType name=«personType»>
    <xs:sequence>
      <xs:element ref=«name»/>
      <xs:element ref=«address» minOccurs=«0»/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name=«nameType»>
    <xs:sequence>
      <xs:element ref=«givenname» maxOccurs=«unbounded»/>
      <xs:element ref=«surname»/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name=«givennameType»>
    <xs:restriction base=«xs:token»/>
  </xs:simpleType>
  <xs:simpleType name=«surnameType»>
    <xs:restriction base=«xs:string»/>
  </xs:simpleType>
  <xs:complexType name=«addressType»>
    <xs:sequence>
      <xs:element ref=«company»/>
      <xs:element ref=«email» minOccurs=«0»/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name=«companyType»>
    <xs:restriction base=«xs:string»/>
  </xs:simpleType>
  <xs:simpleType name=«emailType»>
    <xs:restriction base=«xs:string»>
      <xs:pattern value=«.*@.*\..*»/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

7.3 Tranche de salami (Salami Slice)

Le schéma en tranches de salami définit tous les éléments comme globaux, mais utilise chaque fois pour leur définition des types locaux. Ainsi, les éléments sont réutilisables comme composantes, mais pas les types. Ce concept doit son nom au fait que les définitions des différents éléments sont placées les unes après les autres comme des tranches de salami.

```
<xs:schema xmlns:xs=«http://www.w3.org/2001/XMLSchema»>
  <xs:element name=«person»>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref=«name»/>
        <xs:element ref=«address» minOccurs=«0»/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name=«name»>
    <xs:complexType>
```

```
      <xs:sequence>
        <xs:element ref=«givenname» maxOccurs=«unbounded»/>
        <xs:element ref=«surname»/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name=«givenname» type=«xs:token»/>
  <xs:element name=«surname» type=«xs:string»/>
  <xs:element name=«address»>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref=«company»/>
        <xs:element ref=«email» minOccurs=«0»/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name=«company» type=«xs:string»/>
  <xs:element name=«email»>
    <xs:simpleType>
      <xs:restriction base=«xs:string»>
        <xs:pattern value=«.*@.*\..*»/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```

7.4 Stores vénitiens (Venetian Blinds)

Cette dernière variante est une permutation qui renverse pour ainsi dire la structure en tranches de salami: au lieu de se concentrer sur les éléments en tant que composantes réutilisables, elle définit systématiquement tous les types de manière globale et les éléments localement à l'intérieur de ceux-ci. Il n'y existe par conséquent qu'un seul élément défini globalement, à savoir le Document Element.

```
<xs:schema xmlns:xs=«http://www.w3.org/2001/XMLSchema»>
  <xs:element name=«person» type=«personType»/>
  <xs:complexType name=«personType»>
    <xs:sequence>
      <xs:element name=«name» type=«nameType»/>
      <xs:element name=«address» type=«addressType» minOc-
curs=«0»/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name=«nameType»>
    <xs:sequence>
      <xs:element name=«givenname» type=«givennameType»
                  maxOccurs=«unbounded»/>
      <xs:element name=«surname» type=«surnameType»/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name=«givennameType»>
    <xs:restriction base=«xs:token»/>
  </xs:simpleType>
  <xs:simpleType name=«surnameType»>
    <xs:restriction base=«xs:string»/>
  </xs:simpleType>
  <xs:complexType name=«addressType»>
```

```
    <xs:sequence>
      <xs:element name=«company» type=«companyType»/>
      <xs:element name=«email» type=«emailType» minOccurs=«0»/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name=«companyType»>
    <xs:restriction base=«xs:string»/>
  </xs:simpleType>
  <xs:simpleType name=«emailType»>
    <xs:restriction base=«xs:string»>
      <xs:pattern value=«.*@.*\..*»/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

7.5 Discussion des variantes

Dans la pratique, il arrive fréquemment qu'aucun de ces modèles de conception ne soit appliqué à cent pour cent. Il est toutefois important de connaître ces variantes et d'être conscient de leurs conséquences, notamment en relation avec la question de savoir si un schéma sera réutilisé comme base d'un schéma reposant sur lui ou comme fondement d'un développement ultérieur. Dans les deux cas, il est important de tenir compte du fait que non seulement les éléments et les types sont reliés dans leur forme la plus simple (c'est-à-dire comme types autonomes utilisés comme base pour les éléments ou les attributs), mais aussi que XML Schema autorise entre eux toute une série d'autres relations (décrites au chapitre 7), qui deviennent notamment intéressantes au niveau de l'interaction entre plusieurs schémas.

7.5.1 Recommandations

- **SHOULD**: Le modèle de conception le plus utilisé devrait être la structure en stores vénitiens, avec une définition globale de tous les types et une définition locale ou globale, suivant les besoins, des éléments et des attributs (définition globale uniquement s'ils sont réutilisés à différents endroits). Les types correspondent donc à l'aspect d'un schéma qui est mis spécialement en évidence et qui est bien approprié à la réutilisation grâce à la définition globale générale.
- **SHOULD NOT**: Dans la définition locale d'éléments ou d'attributs, on devrait veiller à ce qu'aucun élément ni attribut de même nom ne soit défini localement parce que cela pourrait perturber les utilisateurs du schéma. Quand il s'agit du même élément, une définition globale devrait être faite, puis référencée. Par contre, on ne devrait pas donner le même nom à des concepts différents.
- **MAY**: Si des éléments doivent réutilisés, ils peuvent aussi être définis globalement (cela est p. ex. indispensable pour les récursions). On tiendra toutefois compte du fait que, dans la structure en stores vénitiens, les types représentent le genre principal de réutilisation, de sorte que chaque cas concerné devrait être considéré séparément.

- **SHOULD**: Si un type doit être réutilisé en relation avec des Identity Constraints (comme décrit au paragraphe 9.5), cette réutilisation devrait être mentionnée clairement dans le type lui-même. La meilleure façon de la documenter est de définir, pour ce type, un élément exemple contenant les Identity Constraints. Dans le type, on peut alors indiquer par un commentaire que, en cas de réutilisation du type, les Identity Constraints doivent être copiés depuis l'élément exemple.

8 Substitution de types

Comme décrit au chapitre 6, XML Schema a créé, avec le niveau des types, un échelon de modélisation entièrement nouveau pour la description et le traitement de documents XML. C'est pourquoi il existe deux manières de considérer un document validé par un schéma XML, d'abord sous l'angle des noms, en considérant les noms d'éléments et d'attributs visibles dans le balisage, et ensuite sous l'angle des types, en considérant les types qui ont été constatés lors de la validation pour ces éléments et attributs.

Les applications travaillant avec XML Schema et XML ont en principe le choix de le faire sous l'angle des noms ou sous celui des types. Le paragraphe 8.1 décrit les conséquences de ces deux manières de procéder. XML Schema offre des mécanismes permettant de rompre l'attribution rigide des noms et des types. Décrits au paragraphe 8.2, ces mécanismes offrent quelques possibilités intéressantes de modélisation, mais compliquent la compréhension du schéma et rendent un peu plus exigeant le travail avec les instances.

8.1 Traitement basé sur les noms contre traitement basé sur les types

Nous discutons dans ce chapitre ce qu'il peut exister comme relations entre les composantes de XML Schema au-delà de la simple relation élément-type (comme discuté au chapitre 7). Il est essentiel de comprendre que toutes ces relations sont autorisées par défaut et que si l'on ne veut pas les utiliser ou interdire leur utilisation aussi pour des applications futures du schéma (pourquoi on veut le faire et pourquoi on devrait même souvent le faire est expliqué au paragraphe **Fehler! Verweisquelle konnte nicht gefunden werden.**), il faut le faire explicitement. Les relations suivantes sont intéressantes dans ce cadre:

- *Utilisation des types.* Les types sont utilisés comme base pour les éléments ou les attributs. Certains types ne doivent toutefois pas être utilisés directement pour des éléments ou des attributs, mais uniquement comme base pour d'autres types dérivés. Dans un tel cas, on peut déclarer un type comme `abstract`, ce qui signifie qu'il ne peut pas être utilisé pour des éléments ou des attributs. En pratique, cela veut dire qu'une instance d'un tel type ne doit jamais apparaître dans un document.
- *Dérivation de type.* Dans XML Schema, la dérivation de type englobe l'extension et la restriction de type pour les types complexes ainsi que la restriction de type, les listes et les unions pour les types simples. La dérivation de type peut être interdite de manière générale par la déclaration `final`. Il est aussi possible d'interdire uniquement des genres spéciaux, de sorte qu'il soit possible p. ex. de déclarer un type qui, bien que restreint, ne puisse pas être élargi.
- *Groupes de substitution.* Les *Substitution Groups* de XML Schema constituent un mécanisme spécial permettant d'utiliser un élément à la place d'un autre (qui doit avoir un type dérivé et se trouver dans le groupe de substitution de l'élément à remplacer). Une telle substitution peut être utile pour les listes hétérogènes, mais nécessite une programmation claire pour le traitement qui doit se faire sur la base des types, car le Content Model de l'élément dans lequel un élément d'un groupe de substitution a été utilisé ne permet pas de voir que d'autre éléments peuvent aussi se

présenter. Si l'on veut éviter le travail supplémentaire pour la prise en charge des groupes de substitution lors de l'implémentation, il faudrait interdire d'emblée ce mécanisme par la déclaration `block` et/ou `final`. Si au contraire l'on veut utiliser des groupes de substitution, il peut être judicieux d'indiquer certains éléments dans des Content Models, mais d'ajouter un `abstract=«true»` dans la déclaration de ces éléments pour forcer leur remplacement par un élément d'un groupe de substitution.

- *Substitution de types.* Les groupes de substitution commentés ci-dessus permettent d'utiliser un élément à la place d'un autre. Cela doit être explicitement autorisé dans le schéma. Par *Type Substitution* (qui utilise l'attribut `xsi:type` dans le document), on peut obtenir un effet similaire en indiquant explicitement le type d'un élément dans le document. Comme il s'agit ici (au niveau des types, qui est important pour XML Schema) du même mécanisme que celui du groupe de substitution, les deux mécanismes peuvent être commandés par le même attribut dans le schéma (`block=«substitution»`).

Comme on le voit d'après ces explications, de nombreuses fonctionnalités de XML Schema sont définies au niveau des types. Comme mentionné au paragraphe 6.1, on constate toutefois aujourd'hui (et cela devra au moins être pris en compte pendant une longue période encore) qu'une grande partie du logiciel traitant le XML n'est pas programmée proprement sur la base des types, ce qui est aussi compréhensible au vu de l'absence d'API normalisés, mais qu'il interprète simplement les noms d'éléments ou d'attributs. Ce dilemme ne peut en fait être résolu que de deux façons:

- *Renoncement au traitement basé sur les types.* On interdit alors dans le schéma toutes les choses se référant au niveau des types (en tout cas la substitution de types décrite au paragraphe **Fehler! Verweisquelle konnte nicht gefunden werden.** et, par là même, aussi les groupes de substitution). On réduit ainsi fortement les possibilités d'erreurs pour les programmeurs «naïfs». Toutefois, le schéma doit alors être conçu de manière qu'un traitement univoque soit possible sans informations de types (ce qui est p. ex. important dans le cas des types `union` pour lesquels l'information de type doit entre autres renseigner sur la manière d'interpréter une valeur déterminée).
- *Traitement explicitement basé sur les types.* Aux endroits du schéma où la substitution de types est autorisée (c'est-à-dire partout sauf indication contraire!), les programmeurs doivent partir du principe que cette substitution est utilisée dans cette version du schéma ou le sera dans les versions suivantes et concevoir le logiciel en conséquence. Cela implique un travail supplémentaire pour les programmeurs, mais offre en revanche une meilleure flexibilité en ce qui concerne le développement ultérieur du schéma.

8.1.1 Recommandations

- **MUST**: Si un schéma XML applique des mécanismes obligeant, lorsque l'on travaille avec des instances, d'utiliser des informations de types (c'est-à-dire les mécanismes décrits au paragraphe 8.2), on l'indiquera clairement dans le schéma ainsi que dans la documentation.

8.2 Mécanismes utilisés dans XML Schema

XML Schema est orienté types, c'est-à-dire que tous les éléments et attributs y ont un type qui est visible dans leur définition, même si on ne le reconnaît pas en considérant seulement une instance (c'est-à-dire le document sans le schéma). XML Schema définit des méthodes de substitution de types pouvant être utilisées de deux manières différentes et permettant d'influencer l'attribution de types à des éléments. Dans les deux cas, ces méthodes ne peuvent être utilisées que pour les éléments.

Dans les deux cas, un type est utilisé dans l'instance en lieu et place du type indiqué dans le schéma, le type se trouvant dans l'instance devant être dérivé du type indiqué dans le schéma. Dans le cas de l'attribut `xsi:type` (paragraphe 8.2.1), le type remplacé est explicitement indiqué dans l'instance; dans les cas des groupes de substitution (paragraphe 8.2.2), le type remplacé découle de l'élément utilisé.

- L'utilisation de l'un des deux mécanismes de substitution (`xsi:type` ou groupes de remplacement) doit être exclusive. Un mélange des deux styles rendrait le schéma difficile à comprendre et en compliquerait inutilement l'utilisation. Cela n'est pas une restriction, car les deux mécanismes assurent la même fonction de base, à savoir permettre d'utiliser dans un modèle de contenu d'autres types que celui de l'élément qui y est mentionné.

Sous l'angle de la conception du schéma, la substitution de types offre de nouvelles possibilités et constitue donc un mécanisme intéressant. D'autre part, les applications programmées «naïvement» peuvent avoir des problèmes avec les mécanismes de la substitution de type, car ils nécessitent un traitement correctement basé sur les types, mais sont souvent encore programmés de manière purement basée sur les noms (plus d'informations à ce propos au paragraphe 8.1). C'est pourquoi la substitution de types ne devrait être utilisée que si les conséquences pour le traitement sont claires et que l'on puisse garantir que le traitement se fasse toujours sur la base des types.

8.2.1 L'attribut xsi:type

Dans une instance, un élément peut indiquer explicitement son type par l'attribut `xsi:type`, ce qui signale que l'élément a à cet endroit un autre type que celui qui est mentionné dans le schéma.

8.2.1.1 Exposé du problème

Si un élément du schéma n'est pas défini explicitement par `block=«...»` ou que l'attribut `blockDefault` de l'élément `xs:schema` soit déclaré, la substitution de types au moyen de l'indication `xsi:type` est autorisée pour cet élément (à supposer qu'il existe un type dérivé du type de l'élément et dont le genre de dérivation n'a pas été interdit par la déclaration `block`).

8.2.1.2 Recommandations

- **SHOULD NOT**: Les mécanismes de la substitution de types `xsi:type` ne devraient jamais être utilisés, sauf pour des raisons importantes. Ils rendent plus difficiles le traitement et la compréhension du schéma ainsi que des instances définies par lui.
- **MUST**: S'il est déclaré explicitement dans le schéma que le mécanisme `xsi:type` peut être utilisé ou même que la conception du schéma exige qu'il le soit (types définis comme `abstract=«true»`), cette utilisation est autorisée, mais doit être documentée de manière adéquate. On veillera ensuite à ce que le traitement de ces instances soit basé sur les types, de manière que l'attribution de type ne s'effectue pas via le type d'un élément du schéma, mais via le `xsi:type` de l'instance.
- **MUST**: Si l'on utilise des mécanismes de substitution de types, on le documentera de manière claire et adéquate, de sorte que les utilisateurs du schéma soient bien informés de cet aspect de ce dernier.

8.2.2 Groupes de substitution

Dans une instance, un élément peut indiquer explicitement, à l'aide de l'attribut `substitutionGroup`, qu'il peut remplacer l'élément référencé dans une instance. Il peut alors apparaître dans une instance partout où l'élément référencé dans l'attribut `substitutionGroup` est autorisé. Ce mécanisme peut aussi être défini pour plusieurs niveaux.

8.2.2.1 Exposé du problème

Si un élément du schéma n'est pas défini explicitement par `final=«...»` ou que l'attribut `finalDefault` de l'élément `xs:schema` soit déclaré, il peut servir de tête d'un groupe de substitution. Les membres de ce dernier référencent la tête dans l'attribut `substitutionGroup` de la définition de l'élément.

8.2.2.2 Recommandations

- **SHOULD NOT**: Sauf raisons contraires importantes, on ne devrait jamais utiliser les mécanismes des groupes de substitution, car ils rendent plus difficiles le traitement et la compréhension du schéma et des instances définies par celui-ci.
- **SHOULD**: S'il est déclaré explicitement dans le schéma que des groupes de substitution peuvent être utilisés ou même que la conception de celui-ci exige qu'ils le soient (des éléments qui sont des Substitution Group Heads sont définis comme `abstract=«true»`), cette utilisation est autorisée, mais doit être documentée de manière adéquate. On veillera ensuite à ce que le traitement de ces instances tienne systématiquement compte des groupes de substitution.
- **MUST**: Si l'on utilise les mécanismes des groupes de substitution, on le documentera de manière claire et adéquate, de sorte que les utilisateurs du schéma soient bien informés de cet aspect de ce dernier.

9 Représentation de structures de données dans le balisage XML

A strictement parler, XML n'est qu'une syntaxe permettant d'échanger des données structurées. La manière de représenter les données d'un champ d'application par des structures XML est laissée au libre choix du développeur du schéma XML. Cela constitue une question très importante, parce que l'on doit le plus souvent travailler longtemps avec les données définies par le développeur du schéma XML et qu'il est donc payant d'être assez soigneux et prévoyant pour représenter les structures de données dans le balisage XML. Les paragraphes ci-après décrivent différents aspects importants pour cette représentation et donnent des recommandations concernant des mécanismes plus ou moins bien adéquats pour résoudre ces problèmes.

9.1 Elément racine (Root Element)

XML Schema ne dispose d'aucune possibilité d'identifier l'élément racine (qui est appelé *Document Element* dans le langage XML standard). C'est pourquoi on ne peut pas savoir d'après un schéma quel est l'élément prévu comme Root Element et qu'il existe, dans la plupart des cas, plusieurs éléments qui pourraient jouer ce rôle. Du point de vue de l'application, il peut aussi être souhaité que plusieurs éléments d'un schéma puissent intervenir comme Root Element, bien que ces éléments soient le plus souvent en petit nombre.

9.1.1 Recommandations

- **MAY**: Les schémas peuvent être conçus de manière que plusieurs éléments interviennent comme éléments racines. Ainsi, un schéma peut servir à valider des instances avec différents éléments racines.
- **SHOULD**: Les éléments racines d'un schéma devraient être identifiés comme tels. Etant donné que XML Schema ne définit aucun mécanisme à cet effet, cette identification doit se faire dans un commentaire désignant les éléments racines potentiels.

9.2 Unités de traitement

XML ne dit absolument rien sur la granularité des données qui sont utilisées par des mécanismes définis par ce langage (les plus importants étant les éléments et les attributs). Il incombe entièrement au concepteur du schéma de décider de quelle manière les contenus doivent être structurés.

Comme règle générale, on peut dire que tout à quoi on doit accéder à partir d'une application devrait être marqué par balisage. Si un schéma est réalisé de cette manière, les applications peuvent utiliser des méthodes simples, basées sur XML (telles que DOM ou XPath), pour traiter les contenus et ne doivent pas rechercher elles-mêmes des structures. Prenons comme exemple de ce problème la question de savoir comment les noms de personnes peuvent être représentés dans le balisage XML.

- `<name>Pieter David van Hoogenband, Sr.</name>`: Dans ce cas, le nom complet est indiqué comme chaîne de caractères dans un élément. Le balisage n'indique pas quelles parties du nom constituent le prénom et le nom de famille. Les applications devraient donc programmer leur propre logique pour le constater, et différentes applications pourraient ainsi parvenir, suivant la logique implémentée, à des résultats distincts.
- `<name given=«Pieter David» family=«van Hoogenband, Sr.»/>`: Dans ce cas, la séparation entre le prénom et le nom de famille est donnée, les deux parties du nom sont clairement séparées et représentées par des attributs différents.
- `<name><given>Pieter David</given><family>van Hoogenband, Sr.</family></name>`: Structurellement, cette variante correspond à la précédente, à la différence que les différentes parties y sont représentées par des éléments et non pas par des attributs. On pourrait être d'avis que le fait que tous les prénoms soient mentionnés dans un élément commun constitue un inconvénient.
- `<name><given>Pieter</given><given>David</given><family>van Hoogenband, Sr.</family></name>`: Dans cet exemple, les prénoms sont marqués séparément par balisage. Par ce schéma, on peut aussi interdire la présence d'un prénom ou fixer le nombre maximal de prénoms autorisés. Reste encore ouverte la question de savoir si la représentation commune des différents éléments du nom de famille est souhaitable.
- `<name><given>Pieter</given><given>David</given><family link=«van» gen=«Senior»>Hoogenband</family></name>`: Si le nom de famille doit être décomposé dans le nom proprement dit et des éléments supplémentaires, cela peut se faire de la manière indiquée ci-dessus. Il est ainsi possible de distinguer simplement et clairement quel est le nom de famille proprement dit et quels sont ses constituants complémentaires.

Laquelle de ces modélisations est utilisée dans un schéma dépend des applications, mais d'une manière générale, pour les noms de personne, une séparation entre le prénom et le nom de famille serait pour le moins souhaitable, car elle est probablement souvent utilisée par les applications. Quant aux approches plus détaillées, on ne peut dire sans connaître le domaine d'application si elles sont souhaitables ou exagérées.

On observera qu'une granularité plus fine donne plus d'indications sur les structures et atteint ainsi une plus grande précision, mais cause toutefois aussi une plus grande restriction, car les données qui ne sont pas adaptées au schéma prédéfini à granularité fine ne peuvent guère y être représentées de manière judicieuse. Dans l'exemple avec les noms de personne, si l'on pose p. ex. l'hypothèse que tous les noms suivent les structures du monde occidental, d'autres noms structurés différemment, p. ex. dans les cultures à noms dynastiques, ne peuvent pas être représentés correctement dans les schémas à structure plus fine, mais peuvent très bien l'être comme chaîne simple de caractères dans le récipient général que constitue un `name` element.

9.2.1 Recommandation

- **SHOULD**: Les unités de traitement qui sont choisies lors de la définition d'un schéma devraient se baser sur la manière dont les données sont structurées du point de vue technique. Leur choix devrait avoir pour objectif de désigner, par balisage XML, les structures importantes pour le traitement, de sorte que les applications n'aient si possible besoin d'aucune méthode extérieure aux technologies XML pour identifier les structures importantes sur le plan technique et pour travailler avec elles.

9.3 Eléments ou attributs?

La décision de choisir des éléments ou des attributs demeure une question récurrente et sans réponse définitive. Deux importantes raisons qui étaient encore en faveur des attributs dans les DTD ne tiennent plus la route lorsque l'on utilise XML Schema:

- *Les attributs ont des types.* Dans les DTD, les attributs ont des types, même si ceux-ci sont très restreints (p. ex. `NMTOKENS`). Cela peut, le cas échéant, constituer un avantage par rapport aux éléments, qui ont comme seul «type» d'autres éléments ou `#PCDATA`.
- *Identity Constraints.* `ID/IDREF` est un type d'attribut spécial permettant les renvois à intégrité référentielle entre les attributs. `ID/IDREF` est utilisé dans de nombreuses applications XML non triviales (le paragraphe 9.5 donne des informations supplémentaires sur ce type d'attribut).

Les deux cas sont couverts par XML Schema pour les éléments également: le concept des *Simple Types* est orthogonal envers les attributs et les éléments, de sorte que tous les types simples peuvent aussi être utilisés pour les éléments, et les Identity Constraints de XML Schema peuvent être utilisées via XPath de la même manière pour les attributs et les éléments (elles offrent en outre des avantages clairs par rapport au type ID/IDREF, comme décrit au paragraphe 9.5).

Sous cet angle, il n'existe, dans XML Schema, plus aucune raison d'utiliser encore des attributs, sauf là où ils sont prescrits de l'extérieur par des normes existantes (p. ex. XML Namespaces ou XLink) ou par d'autres conventions. Comme auparavant, seuls des critères subjectifs, tels que la clarté et la compacité du codage, parlent en faveur des attributs. On tiendra compte également de la normalisation des valeurs d'attribut dans XML (définies par la norme XML elle-même), qui fait que les whitespaces et notamment les fins de ligne sont traités différemment dans les éléments et dans les valeurs d'attribut. Ce comportement peut toutefois être influencé par l'attribut `whiteSpace` pour les types simples (de manière limitée).

9.3.1 Recommandation

- **SHOULD**: Pour la structuration, on devrait toujours utiliser des éléments, car ceux-ci peuvent être complétés de manière simple en cas de besoin, par exemple par des attributs supplémentaires ou par leur subdivision en structures plus fines.

- **MAY**: Les attributs devraient être utilisés avec précaution. Ils sont soumis à des restrictions importantes (pas de répétition, pas de possibilité de structuration plus détaillée) et constituent par conséquent, en cas de modifications ultérieures d'un schéma, un endroit auquel, le cas échéant, les modifications souhaitées ne peuvent plus être exécutées de manière simple.

9.4 Représentation de valeurs vides

Dans un modèle de données, il existe souvent la possibilité que certaines indications soient optionnelles, c'est-à-dire qu'elles ne doivent pas impérativement figurer dans les instances. La question se pose de savoir comment une telle occurrence optionnelle peut être réalisée de la meilleure façon possible dans un schéma XML. XML et XML Schema permettent différents genres de réalisation, que nous décrivons ci-après.

9.4.1 Exposé du problème

Les valeurs nulles peuvent être représentées dans XML et, en particulier, lors de l'utilisation de XML Schema par des méthodes différentes. Pour obtenir une syntaxe applicable de manière générale, on devrait toutefois, dans ce domaine aussi, s'efforcer de respecter certaines normes. Sur le plan technique, les possibilités sont les suivantes:

- Absence des composantes concernées (éléments ou attributs)
    - Avantage: mécanisme simple
    - Inconvénient: les éléments/attributs doivent être déclarés comme optionnels dans le schéma. Cela n'est pas toujours souhaité.
- Contenu vide des composantes concernées (éléments ou attributs)
    - Avantage: mécanisme simple
    - Inconvénient: une composante ne peut pas être vide si une restriction est définie pour elle (donc p. ex. si le type prescrit qu'une chaîne de caractères doit avoir une longueur déterminée).
- Utilisation de l'attribut `xsi:nil=«true|false»` à l'intérieur de l'instance; l'élément (ce mécanisme ne fonctionne pas pour les attributs) doit alors être déclaré dans le schéma avec `nillable=«true»`.
    - Avantages: (1) La valeur `xsi:nil` est définie en plus de l'élément vide, il est donc possible de distinguer une valeur vide d'une valeur absente si cela est souhaité. (2) Cette méthode permet d'avoir un contenu vide pour des éléments qui sont dotés de restrictions, p. ex. un type `xs:string` avec une longueur minimale de 2 caractères.
    - Inconvénients: (1) Cette méthode ne peut être utilisée que pour les éléments. (2) Elle n'est pas souvent utilisée car elle contient des restrictions importantes. (3) Les applications doivent prendre explicitement en charge ce mé-

canisme; si tel n'est pas le cas, des problèmes peuvent survenir lors de l'interprétation.

9.4.2 Recommandation

- **SHOULD**: Les valeurs nulles devraient être représentées au moyen d'éléments ou d'attributs optionnels, c'est-à-dire par l'absence des composantes concernées ou par un contenu vide pour celles-ci.
- **SHOULD NOT**: Le mécanisme `xsi:null` ne devrait pas être utilisé.

9.5 Identification et références

Il est fréquemment nécessaire, dans les modèles de données, d'en identifier les parties structurelles et de les rendre ainsi référençables. La méthode la plus utilisée à cet effet est l'attribution de noms par lesquels peuvent être appelées les structures à identifier et qui peuvent être utilisées dans les références.

Le mécanisme le plus connu pour cela est l'utilisation des attributs `ID/IDREF` des DTD XML, qui permettent d'une part, de donner un nom univoque aux éléments (par un attribut déclaré comme ID) et, d'autre part, de faire référence à des ID existants (par un attribut déclaré comme `IDREF`). Le processeur XML vérifiera alors, lors de la validation, si l'univocité (`ID`) et l'existence (`IDREF`) existent. On peut aussi faire appel à ce mécanisme dans XML Schema si l'on utilise les types `xs:ID` et `xs:IDREF` définis dans [xmlschema2sec].

Le mécanisme `ID/IDREF` comporte deux gros inconvénients qui peuvent fortement restreindre son utilité: l'utilisation des types `ID/IDREF` ne force pas seulement l'univocité ou l'existence des noms dont l'on se sert, mais détermine aussi qu'ils doivent être des noms XML. Il en résulte notamment que les nombres ne sont pas autorisés, ce qui est souvent gênant quand des noms existent déjà et qu'ils sont fournis sous la forme de numéros, p. ex. par une base de données. L'autre inconvénient réside dans le fait que les concepts s'appliquent globalement pour un document, c'est-à-dire que deux différents attributs déclarés comme ID ne peuvent pas prendre la même valeur et que les références `IDREF` ne renvoient pas obligatoirement à un nom du domaine souhaité. Ces deux inconvénients peuvent être évités au moyen des Identity Constraints de XML Schema.

Au vu de ces graves inconvénients du concept `ID/IDREF`, un nouveau mécanisme a été introduit dans XML Schema: les Identity Constraints. Ces dernières sont conçues pour le même domaine d'application que le mécanisme `ID/IDREF`, mais sans en comporter les principaux inconvénients. Elles permettent notamment de choisir librement le type des noms et peuvent se référer spécifiquement à des noms déterminés du schéma (via un XPath), de sorte qu'elles ne sont plus globales. Elles gardent toutefois l'inconvénient qu'elles ne s'appliquent aussi qu'à un seul document et ne peuvent donc pas exprimer de dépendances entre différents documents.

ATTENTION: Bien qu'elles constituent une nette amélioration par rapport aux concepts `ID/IDREF` des DTD, les Identity Constraints sont toutefois, de par l'utilisation de XPaths, liées aux noms des éléments et non pas à leurs types! C'est pourquoi leur utilisation est en

conflit avec la recommandation, formulée au chapitre 8, de préférer le traitement basé sur les types à celui basé sur les noms. Notons en particulier que les Identity Constraints sont contenues dans les déclarations d'élément. Si les types deviennent la construction préférée pour la réutilisation, comme décrit au paragraphe 7.5, les Identity Constraints n'en tiennent pas compte. C'est pourquoi les types destinés à la réutilisation devraient être documentés de manière très claire si leurs éléments devaient contenir une Identity Constraint.

9.5.1 Recommandation

- **SHOULD**: Si l'univocité, l'existence ou le référencement de noms doivent être garantis, on devrait utiliser des Identity Constraints de XML Schema pour définir les contraintes correspondantes. On veillera à définir alors les XPaths de manière aussi restrictive que possible, afin d'éviter tout conflit en cas de modification ultérieure du schéma.
- **SHOULD**: Si des Identity Constraints sont utilisées, les structures auxquelles elles renvoient devraient faire appel à leurs propres types pour les noms. De cette manière, les types pourront être réutilisés de manière simple si les noms correspondants doivent être repris à un autre endroit du schéma.
- **SHOULD NOT**: Le mécanisme DTD des attributs `ID/IDREF` ne devrait pas être utilisé, car il est soumis à de fortes restrictions et ne permet pas de définir les contraintes de manière aussi exacte que cela est possible avec les Identity Constraints.
- **MAY**: Si l'application fixe des exigences (p. ex. univocité ou référencement sur plusieurs documents) qui ne peuvent pas être implémentées par des Identity Constraints de XML Schema, celles-ci peuvent être définies par une identification et des contraintes externes. On utilisera alors des types spécifiques pour les structures touchées par ces définitions externes et on documentera clairement dans le schéma que les valeurs de ces types sont soumises à des contraintes définies en dehors du schéma.
- **SHOULD**: Si des Identity Constraints doivent intervenir dans la réutilisation de types déterminés, on le documentera dans le schéma, de préférence en définissant, pour le type concerné, un élément exemple contenant les Identity Constraints. On pourra alors ajouter un commentaire dans le type pour signaler que, en cas de réutilisation de celui-ci, il faudra copier les Identity Constraints de l'élément exemple.

9.6 Listes de valeurs

Des listes de valeurs font très souvent partie d'un vocabulaire parce que seules des valeurs déterminées, bien définies, doivent être autorisées pour certaines parties du modèle de données. En principe, les listes de valeurs peuvent être définies de deux manières:

- Liste de valeurs statique: la liste de valeurs est définie comme type, et le type définit les valeurs autorisées directement dans une énumération. Cela se fait habituellement au moyen d'un Simple Type adéquat et d'Enumeration Facets permettant d'énumérer les valeurs autorisées.

- Liste de valeurs dynamique: si les valeurs ne doivent pas être définies dans le schéma, le type est défini comme type lexicalement restreint, qui limite le domaine des valeurs possibles, mais n'énumère pas directement ces valeurs. Les valeurs actuelles doivent alors être définies dans une liste gérée extérieurement, et les applications doivent connaître cette liste et y avoir accès afin de pouvoir procéder à une véritable vérification des types.

Des exemples simples et typiques de ces deux approches figurent dans le schéma XML pour XML Schema, qui contient, par exemple, la définition suivante comme liste de valeurs statique:

```
<xs:simpleType name=«derivationControl»>
  <xs:restriction base=«xs:NMTOKEN»>
    <xs:enumeration value=«substitution»/>
    <xs:enumeration value=«extension»/>
    <xs:enumeration value=«restriction»/>
    <xs:enumeration value=«list»/>
    <xs:enumeration value=«union»/>
  </xs:restriction>
</xs:simpleType>
```

Dans ce cas, les valeurs autorisées, qui définissent les valeurs admises pour la dérivation de types dans XML Schema, sont indiquées directement dans le schéma. Cela est judicieux parce que ces valeurs ne seront guère modifiées ou, en cas de modification, XML Schema devrait de toute façon être remanié de manière si fondamentale qu'il en résulterait un schéma entièrement nouveau.

A l'opposé, il existe aussi dans le schéma XML pour XML Schema, des exemples de listes de valeurs dynamiques, dans lesquels le schéma renvoie à des listes externes:

```
<xs:simpleType name=«language» id=«language»>
 <xs:restriction base=«xs:token»>
  <xs:pattern value=«[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*»>
   <xs:annotation>
    <xs:documentation source=«http://www.ietf.org/rfc/rfc3066.txt»>
      pattern specifies the content of section 2.12 of XML 1.0
      and RFC 3066 (Revised version of RFC 1766).
    </xs:documentation>
   </xs:annotation>
  </xs:pattern>
 </xs:restriction>
</xs:simpleType>
```

Dans cet exemple pour le type `language` de XML Schema, les valeurs admises ne sont pas directement définies dans le schéma, qui définit uniquement un schème restreignant lexicalement l'ensemble des valeurs admises. Les valeurs concrètes pour ce type doivent être lues d'une liste externe, laquelle est définie par un document IETF.
La raison de cette construction est ici que la liste des langues, d'une part, est une liste en cours de développement et que, d'autre part, les développeurs de XML Schema ne voulaient pas indiquer les langues qui seraient définies dans XML Schema et celles qui ne le seraient pas. Sur le plan formel, le schéma autorise donc des marquages de langue tels que `xx-YY` (qui serait validé sans problème), et c'est seulement la consultation du document [RFC3066]

ou de la liste correspondante auprès de la IANA qui indiquerait que ce marquage n'est pas un marquage de langue valable.

9.6.1 Recommandation

- **SHOULD**: Si les listes de valeurs sont des listes statiques dont les valeurs sont connues de manière exhaustive et resteront stables en principe, celles-ci devraient être énumérées dans le schéma au moyen d'un Simple Type et d'Enumeration Facets. Pour mieux gérer ces listes des valeurs et pouvoir procéder à de meilleurs contrôles d'accès, il est en général judicieux de les externaliser dans un document de schéma autonome (voir paragraphe 11.1).
- **SHOULD**: Si les listes des valeurs sont des listes dynamiques dont les valeurs ne sont pas connues de manière exhaustive ou peuvent être modifiées, elles ne devraient être définies que par une restriction lexicale la plus exacte possible et par la référence à une liste externe. De nouvelles valeurs peuvent ainsi être ajoutées aux listes des valeurs sans qu'il faille modifier quelque chose au schéma.

9.7 Marquage de la langue des contenus

Dans les documents comprenant des contenus en langue naturelle, une présentation multilingue de ceux-ci est souvent exigée. La précision requise (p. ex. si exactement un contenu est autorisé dans une langue à indiquer ou si des contenus parallèles en plusieurs langues sont admis) est une question à régler dans les spécifications de l'application.

Indépendamment de la conception spécifique des contenus, les marquages de la langue devraient toutefois toujours être réalisés à l'aide de l'attribut `xml:lang` qui est défini dans la norme XML elle-même. Comme il provient de l'espace nominatif XML lui-même (donc de l'espace nominatif que la norme XML Namespaces réserve pour XML), cet attribut doit être intégré par importation (comme décrit au paragraphe 11.2) d'un schéma qui lui est propre. Un schéma XML pour l'attribut `xml:lang` est très simple. En voici un exemple:

```
<xs:schema xmlns:xs=«http://www.w3.org/2001/XMLSchema»
           targetNamespace=«http://www.w3.org/XML/1998/namespace»>
  <xs:attribute name=«lang» type=«xs:language»/>
</xs:schema>
```

Si par contre le choix des langues devait être restreint, il serait aussi possible, comme décrit au paragraphe 9.6, d'énumérer dans le schéma `xml:lang` l'ensemble des langues autorisées.

9.7.1 Recommandation

- **SHOULD**: Si des marquages de la langue sont nécessaires pour des contenus, ils devraient être réalisés dans un attribut `xml:lang`, car celui-ci est défini dans la norme XML elle-même et constitue une convention générale pour les marquages de

la langue. [eCH-0050] définit un schéma correspondant, qui est mis à disposition pour réutilisation dans le cadre de eCH.

10 Versions des schémas XML

Au fil du temps, les schémas se modifient, car ils doivent être adaptés aux nouveaux développements et conditions cadres. Il faut donc se poser la question des répercussions à craindre, en cas de modification d'un schéma, pour les applications se basant sur des versions plus anciennes de celui-ci. Si un schéma est conçu de manière ouverte (comme décrit au paragraphe 13.1), ces extensions restent sans conséquences à condition qu'elles utilisent les points prévus à cet effet.

10.1 Exposé du problème

Dans ce contexte, une question générale se pose en cas de modification d'un schéma: peut-on encore parler d'une nouvelle version ou les changements affectant le schéma sont-ils tels qu'il serait plutôt indiqué de parler d'un nouveau schéma.

Dans l'hypothèse de la création d'une nouvelle version, il faut se demander quelles modifications devraient donner lieu à une version mineure et lesquelles à une version majeure. Comme directives générales, on peut appliquer les principes suivants:

- Si la sémantique de certaines constructions est modifiée, il y a toujours lieu de définir une nouvelle version majeure, de sorte que le traitement de ces constructions doive s'adapter à la nouvelle sémantique. Si un destinataire utilise une nouvelle version de schéma qui ne lui permettrait plus de valider d'anciens documents, il y a aussi lieu de définir une nouvelle version majeure.
- Pour les autres modifications, donc celles dans lesquelles d'anciennes instances peuvent être validées avec le nouveau schéma, une version mineure suffit.

Dans une considération plus détaillée, les observations suivantes peuvent être faites en ce qui concerne les modifications du schéma:

- S'il existe des points d'extension et si les nouvelles structures sont ajoutées à ces endroits-là, seule une nouvelle version mineure est nécessaire.
- Si des parties du schéma qui était optionnelles sont maintenant définies comme obligatoires, une version majeure est nécessaire parce que les applications qui génèrent les instances de ce schéma doivent être adaptées en conséquence.
- Si seule la documentation change, une version mineure suffit (toutefois uniquement à condition que la documentation ne décrive pas une nouvelle sémantique!).
- Si des extensions sont introduites à des endroits où aucun point d'extension n'a été prévu, une nouvelle version majeure doit être générée.

Comme nous l'avons mentionné, en cas de modification massive d'un schéma, il faudrait bien se demander si le résultat peut effectivement encore être considéré comme une nouvelle version de l'ancien schéma ou s'il ne constitue pas plutôt un schéma entièrement nouveau, qui présente seulement certaines correspondances au niveau du contenu avec le schéma initial.

Dans les nouvelles versions d'un schéma, il peut arriver que des valeurs, éléments ou attributs admis auparavant ne doivent plus être autorisés. Au lieu de les interdire, il peut être judicieux de continuer de les admettre dans le nouveau schéma, mais de les marquer clairement comme *deprecated*, de sorte qu'ils restent acceptés pour la validation, mais qu'il soit clair que ces valeurs ou structures devraient être ignorées lors du traitement. Cette méthode permet une meilleure coexistence des applications de différentes versions de schémas.

10.2 Recommandations

- **SHOULD**: Une nouvelle version mineure devrait être générée si les instances suivant l'ancien schéma peuvent être traitées correctement lors de la validation et de l'interprétation selon le schéma modifié.
- **SHOULD**: Une nouvelle version majeure devrait être générée si les instances suivant l'ancien schéma ne peuvent plus être traitées correctement lors de la validation et de l'interprétation selon le schéma modifié.
- **SHOULD NOT**: Si la nouvelle version interdit des valeurs ou structures permises auparavant, cette interdiction ne devrait pas être réalisée dans la définition du schéma, mais être signalée comme *deprecated* au niveau de l'application, de manière que les anciennes instances puissent continuer d'être traitées.
- **MUST**: Pour permettre l'identification de la version mineure d'un schéma et de ses instances (la version majeure étant identifiée par le nom d'espace nominatif), on doit utiliser l'attribut `version` du schéma qui contient uniquement la version mineure. Dans les instances, la version mineure fait l'objet d'un marquage qui doit être prévu dans le schéma lui-même, mais qui devrait de préférence être réalisé par l'utilisation de l'attribut prévu dans [eCH-0050] (cet attribut est aussi recommandé dans [eCH-0018]).
- **MUST**: La même version majeure doit être prise en charge de chaque côté en cas d'échange de données dans les deux sens. Si tel n'est pas le cas, la communication ne peut pas avoir lieu.
- **MAY**: S'il s'agit d'un échange de données essentiellement unidirectionnel, on peut éventuellement en tenir compte lors de la définition de la version et, si une version mineure est utilisée, des modifications peuvent être acceptées tant qu'elles n'entraînent aucun problème de compatibilité du côté du destinataire des données. Il est alors nécessaire de réaliser une bonne documentation, décrivant clairement le scénario et définissant ainsi les cas dans lesquels une communication est possible.
- **MAY**: Les recommandations ci-dessus ne sont applicables que pour les schémas développés et utilisés de manière productive. Une procédure simplifiée est possible pour la phase de développement, avant l'utilisation productive, et pour les schémas non destinés à une telle utilisation.

11 Structuration de schémas et de documents de schéma

Considéré sur le plan logique, un schéma XML est une unité, c'est-à-dire que toutes ses composantes doivent se trouver en cohésion et qu'aucune «référence ouverte» ne peut exister. D'un autre côté, il est possible de structurer les documents de schéma et ceux-ci sont, par opposition aux composantes du schéma, la manière concrète à utiliser pour définir des schémas XML. Nous devons insister encore sur ces deux aspects:

- Un *document de schéma* est un document XML qui utilise comme espace nominatif l'espace nominatif XML Schema (qui devrait être lié au préfixe `xs`) ainsi qu'un élément `xs:schema` de cet espace nominatif comme Document Element et qui définit ensuite différentes composantes à l'intérieur de cet élément, p. ex. des éléments, des attributs ou des types.
- Les *composantes de schéma* sont le modèle de données de XML Schema; il s'agit de concepts abstraits qui sont définis au moyen d'un document de schéma. XML Schema lui-même est défini à l'aide de ces composantes et les documents de schéma constituent une syntaxe XML avec laquelle l'on peut représenter ces composantes.

Partant de ces définitions, il est important d'observer qu'il existe une quantité d'aspects qui sont reconnaissables dans le document de schéma sans toutefois être pertinents pour le modèle de données, véritable important, des composantes de schéma. Pour les auteurs et utilisateurs de XML Schema, ces aspects peuvent bien entendu quand même être pertinents pour des raisons d'organisation, de contrôle d'accès ou de vue d'ensemble, et c'est pourquoi nous allons les aborder ici.

En ce qui concerne la dénomination de documents de schéma et de schémas (donc la collection abstraite des composantes), on peut dire que la dénomination d'un document de schéma est sans importance et que le nom (p. ex. URI) sous lequel un document de schéma est enregistré ne devrait pas être utilisé pour l'identification du schéma. La dénomination du schéma par son nom d'espace nominatif est l'appellation pertinente du schéma et devrait seule être utilisée pour l'identification (ce qui correspond aux directives fixées dans [eCH-0018] concernant l'identification du schéma par des instances). C'est pourquoi les documents de schéma peuvent, p. ex. pour des raisons d'efficience, être copiés sans problème (p. ex. sur des supports de mémoire locaux) pour que l'on puisse y accéder de manière plus efficace. Garantir la cohérence de ces copies et organiser leur gestion est alors du ressort de l'application.

11.1 Structuration de documents de schéma

A l'intérieur d'un document de schéma, l'ordre des composantes globales (donc de celles qui sont enfants de l'élément `xs:schema`) est sans importance; il incombe à l'auteur de choisir l'ordre qui convient à ses besoins. Les relations concernant XML Schema entre ces composantes sont établies uniquement par les noms de celles-ci (le genre des relations possibles est décrit en détail au chapitre 7). Toutes les composantes globales doivent donc porter un nom.

Pour les grands documents de schéma, la quantité des composantes peut rapidement faire croître le schéma à un point tel qu'il devienne peu clair. Pour pallier cette difficulté, on devrait utiliser l'élément `xs:include` afin de modulariser les documents de schéma. L'utilisation de `xs:include` n'a aucune conséquence sur les composantes de schéma définies, mais peut nettement améliorer l'organisation, le contrôle d'accès et la vue d'ensemble lors de l'utilisation de XML Schema (en outre, les documents de schéma modularisés, comme décrit au paragraphe 11.2, peuvent être réutilisés dans différents scénarios).

L'élément `xs:include` de XML Schema fonctionne de manière très simple: on peut pratiquement considérer que le contenu textuel du document de schéma référencé est intégré en lieu et place de l'élément `xs:include` (naturellement sans tenir compte de l'enveloppe constituée par l'élément `xs:schema` du document de schéma intégré). Cette manière de voir n'est pas tout à fait exacte, parce que les déclarations d'espace nominatif sont résolues au préalable et qu'une certaine interprétation est donc faite avant l'intégration, mais ce ne sont que des détails. `xs:include` fonctionne de manière récursive, c'est-à-dire que l'intégration peut être imbriquée autant que l'on veut, et que toutes ces dépendances doivent être résolues avant l'interprétation du schéma.

Avec `xs:redefine`, XML Schema offre un mécanisme allant au-delà de `xs:include.` En effet, les documents de schéma référencés par `xs:redefine` font dès lors aussi, comme avec `xs:include`, partie intégrante du document qui les inclut, mais ce mécanisme autorise en plus de remplacer de manière ciblée des composantes du schéma supérieur. Il est ainsi possible de réutiliser des documents de schéma, mais en les modifiant à des endroits déterminés.

Tant `xs:include` que `xs:redefine` comportent des restrictions importantes en ce qui concerne l'espace nominatif utilisé: le document de schéma intégré doit avoir le même `targetNamespace` que le document de schéma intégrant (les généralités sur l'utilisation d'espaces nominatifs dans XML Schema figurent au chapitre 5).

En outre, il est permis qu'un schéma intégré ne définisse aucun `targetNamespace`, les composantes du schéma intégré prenant alors l'espace nominatif du schéma intégrant. Les composantes d'un document de schéma sans `targetNamespace` pouvant ainsi se présenter sous différents noms d'espace nominatif, ce genre d'utilisation est aussi appelé *schéma caméléon* (les composantes intégrées prennent le nom d'espace nominatif de l'environnement dans lequel elles sont utilisées).

11.1.1 Exposé du problème

Il existe de très nombreuses manières de répartir les définitions de schéma sur les documents de schéma. L'auteur d'un schéma devrait tenir compte, lorsqu'il définit ce dernier, non seulement de ses propres besoins, mais aussi de ceux des utilisateurs du schéma. Comme on peut le dire de manière générale pour la modularisation, il faut rechercher le compromis idéal entre des définitions monolithiques très grandes et difficiles à manier, d'une part, et une multitude complexe de très petits modules, d'autre part.

Les schémas caméléons constituent un problème particulier car, malgré leur attrait (réutilisation simple de composantes dans des contextes différents), ils comportent aussi un potentiel

de risque. Les deux plus grands problèmes pouvant résulter de l'utilisation de schémas caméléons sont les confusions de nom (name clashes) parce que des composantes réutilisées ne sont pas clairement séparées par le nom d'espace nominatif ainsi que l'absence de possibilité de reconnaître les points communs entre les composantes, car celles-ci apparaissent chaque fois sous un autre nom d'espace nominatif et que leur origine commune n'est plus reconnaissable.

11.1.2 Recommandations

- **SHOULD**: Pour des raisons de modularisation, les documents de schéma dépassant une certaine taille devraient être structurés par l'utilisation de `xs:include`. Cela permet de réaliser des schémas plus clairs ainsi que d'en réutiliser des parties (p. ex. des listes de valeurs, comme décrit au paragraphe 9.6).
- **SHOULD NOT**: Le mécanisme `xs:redefine` de XML Schema ne devrait pas être utilisé, parce qu'il cause des dépendances compliquées entre différents documents de schéma et, surtout, qu'il peut provoquer, en cas de modification ultérieure de documents de schéma, des erreurs difficiles à découvrir.
- **SHOULD NOT**: Les schémas caméléons (c'est-à-dire les documents de schéma sans `targetNamespace`, qui prennent par conséquent l'espace nominatif du document de schéma dans lequel ils sont intégrés) ne devraient pas être utilisés, car ils comportent diverses possibilités d'erreurs et ne signalent pas les points qu'ils ont en commun.
- **MAY**: La modularisation d'un schéma peut aussi être utile si plusieurs groupes d'une équipe travaillent au développement ou à l'extension de celui-ci. Elle sera alors conçue en fonction de l'attribution, au niveau du contenu, des différentes parties du schéma aux différents groupes de l'équipe.

11.2 Structuration de schémas

La structuration, expliquée au paragraphe 11.1, de documents de schéma détermine comment devraient être structurés les différents documents de schéma définissant un schéma. En plus des mécanismes que nous avons décrits plus haut, XML Schema offre le mécanisme `xs:import` qui, à la différence de `xs:include`, ne référence pas des documents de schéma, mais d'autres schémas. Avec `xs:import`, des schémas existants sont donc référencés dans un schéma, et la séparation a lieu via l'espace nominatif, c'est-à-dire que le schéma importé doit avoir un autre espace nominatif que le schéma qui l'importe et que les composantes des deux schémas peuvent ainsi être distinguées les unes des autres.

Lors de la définition de documents de schéma, on devrait considérer que ceux-ci peuvent être utilisés tant via `xs:include` que via `xs:import`. Ainsi, une liste de valeurs (voir paragraphe 9.6) définie pour un projet dans un document de schéma spécifique et utilisée via `xs:include` pourrait être importée à un moment ultérieur par un autre schéma via `xs:import`, de sorte que le type définissant la liste de valeurs devienne réutilisable dans ce nouveau schéma comme type du schéma importé.

11.2.1 Exposé du problème

Lors de la définition de documents de schéma et de schémas, la forme sous laquelle un schéma ou des parties de celui-ci pourrait être réutilisé n'est pas entièrement prévisible. On veillera par conséquent (comme déjà décrit au paragraphe 11.1.1) à réaliser de manière modulaire la définition de schémas et de documents de schéma, de sorte à garder le plus ouvertes possible les possibilités de réutilisation ultérieure.

11.2.2 Recommandations

- **SHOULD**: Dans la mesure du possible, des solutions déjà disponibles pour des aspects partiels d'un nouveau schéma devraient pouvoir y être réutilisées par importation des schémas concernés. Une telle possibilité facilite le développement de schémas et constitue aussi une aide précieuse pour les utilisateurs de schémas, car ils se servent ainsi de schémas déjà connus et peuvent donc réutiliser du savoir-faire et du code déjà existants.
- **SHOULD**: Si plusieurs équipes ou projets participent au développement, ils ne devraient pas réaliser un schéma commun, mais plusieurs schémas distincts. Cette manière de faire permet de mieux découpler les différentes parties du développement et d'éviter les problèmes que pourrait causer une progression plus ou moins rapide dans les équipes ou projets concernés.

12 Implémentation de relations référentielles

Les modèles de données contiennent souvent des dépendances qui doivent être représentées dans leurs implémentations. De par sa nature hiérarchique, XML connaît deux possibilités d'implémenter ces dépendances:

- *Les hiérarchies*: dans XML, ce genre de relations est très simple et naturel à comprendre, car les documents XML sont toujours des arborescences Les structures hiérarchiques font donc partie du modèle de base de XML. Mais comme il s'agit d'arborescences, les limites inhérentes à ce genre de structure doivent être prises en considération, notamment la restriction signifiant que seules les relations 1:n peuvent être représentées de cette manière. Les relations n:m ne peuvent pas être représentées sans autre sur des hiérarchies. Une autre restriction réside dans le fait qu'une seule relation 1:n peut être représentée, car un élément ne peut être contenu que dans un autre.
- *Les références*: pour contourner les restrictions mentionnées ci-dessus, il faut choisir dans XML une implémentation utilisant non pas des hiérarchies, mais des références. Les structures à mettre en relation les unes avec les autres sont alors toutes identifiées, et la relation est établie par le fait que les références sont mises en relation les unes avec les autres. Ce principe est très similaire à celui des clés et des clés de tiers dans les bases de données. Pour que l'on puisse tirer des conclusions sur les relations autorisées, il existe dans XML des mécanismes complémentaires, qui sont les types d'attribut `ID/IDREF` dans les DTD et les Identity Constraints décrites au paragraphe 9.5 dans XML Schema.

Partant de ces différentes solutions, on peut remarquer qu'il faut faire appel aux références pour les relations référentielles complexes (n:m ou plus d'une relation 1:n), alors qu'une représentation par des hiérarchies est aussi possible pour les relations 1:n simples.

XML lui-même ainsi que les fonctionnalités des différents langages de schéma XML se concentrent plutôt sur la définition des relations hiérarchiques autorisées. Il existe certes des mécanismes pour définir les conditions à respecter pour les références (les types d'attribut `ID/IDREF` des DTD, mentionnés ci-dessus, et les Identity Constraints de XML Schema, décrites au paragraphe 9.5), mais ceux-ci n'offrent aucune possibilité particulièrement appropriée pour définir les conditions à respecter pour les références. Si des références sont utilisées, de nombreuses conditions résultant du modèle spécifique ne peuvent donc pas être définies dans le schéma XML, mais seulement être documentées, ou alors définies à l'aide de l'un des mécanismes décrits au chapitre 14.

En cas d'utilisation de relations hiérarchiques ou référentielles dans XML Schema, on devrait réfléchir aux aspects suivants:

- *Pour la hiérarchie, contre les références:* si des documents sont principalement destinés à l'utilisation par des technologies XML (p. ex. Xpath, ou XSLT et Xquery qui en découlent), celles-ci peuvent nettement mieux travailler sur des données structurées hiérarchiquement que sur des documents de structure plate et systématiquement non hiérarchique.

- *Pour les références, contre la hiérarchie:* les relations hiérarchiques sont moins robustes par rapport aux modifications structurelles dans les instances, des modifications minimes au niveau des données pouvant, dans certaines circonstances, entraîner des changements relativement importants dans les hiérarchies, car des «sous-arborescences» entières peuvent devoir être déplacées à l'intérieur du document.

A part les réflexions de principe sur le pour et le contre des structures hiérarchiques ou référentielles, il importe aussi de se demander comment sont construites les applications travaillant avec les structures. On ne devrait toutefois jamais oublier que la durée de vie des schémas et des données qu'ils décrivent dépasse souvent celle des applications correspondantes et que l'on ajoutera probablement plus tard de nouvelles applications qui ne devront pas obligatoirement respecter les mêmes conditions que les applications actuelles. C'est pourquoi une grande prudence est de mise avant de procéder à une modélisation trop fortement basée sur les applications actuelles.

12.1 Recommandations

- **MAY**: S'il est judicieux, pour des raisons concernant l'implémentation, de renoncer aux représentations hiérarchiques (parce que, par exemple, la production et la consommation des données passent toujours par un gestionnaire de bases de données relationnelles), il est autorisé de renoncer à la représentation hiérarchique des données. On ne devrait toutefois le faire qu'avec attention et précaution, parce que les contraintes pourraient être modifiées pour des raisons d'implémentation (p. ex. si l'on passe à des bases de données XML) et que le schéma ne serait alors plus adapté au nouvel environnement.
- **SHOULD**: Si des références sont utilisées, les contraintes auxquelles elles sont soumises devraient être décrites aussi bien que possible à l'aide d'Identity Constraints (avec la réserve faite au paragraphe 9.5) et les contraintes supplémentaires devraient être documentées.
- **SHOULD**: Si des schémas sont conçus essentiellement pour l'utilisateur final, p. ex. dans le cas de documents XML qui sont utilisés (étudiés ou édités) par des personnes, tels que des fichiers de configuration ou des documents orientés Web, la modélisation aux endroits concernés devrait plutôt être hiérarchique. Les structures hiérarchiques sont plus simples à comprendre pour l'être humain que celles qui sont reliées entre elles par de nombreuses références.
- **SHOULD**: Si des schémas sont conçus essentiellement pour le traitement par ordinateur, le modèle peut être défini de manière plus plate que pour le cas essentiellement destiné à l'utilisateur final. Aux endroits où existe une structure hiérarchique dûment justifiée, inhérente au modèle de base, une telle structure devrait toutefois aussi être définie dans le schéma XML.

13 Schémas ouverts et évolutifs

Si l'on veut pouvoir procéder à l'extension judicieuse d'un schéma, on tiendra compte de deux aspects distincts: d'abord, la conception du schéma de départ doit d'emblée être réalisée de manière à autoriser une extension. Nous avons déjà considéré les aspects importants à cet égard quand nous avons étudié, entre autres, si les types, éléments et attributs devaient être déclarés localement ou globalement. C'est seulement s'il sont définis globalement qu'ils portent un nom et qu'ils peuvent être référencés à des fins de redéfinition. Deux autres aspects, non mentionnés jusqu'ici, du même domaine sont les *Named Groups* (XML Schema connaît les *Attribute Groups* et les *Model Groups*), qui permettent de définir globalement d'autres composantes d'une définition de type et de les rendre ainsi redéfinissables. Un schéma se basant systématiquement sur des composantes globales devient, il est vrai, assez volumineux, mais offre de meilleures possibilités pour réutiliser ou redéfinir des composantes.

Le deuxième aspect, qui va en fait au-delà de la question directe de l'évolutivité du schéma de départ, est la mesure dans laquelle les documents autorisent les extensions, donc p. ex. la mesure dans laquelle des attributs ou des éléments sont autorisés à des endroits auxquels peuvent apparaître ensuite des informations supplémentaires dans des extensions du schéma. Nous voulons considérer brièvement cet aspect d'ouverture d'un schéma, qui est en principe entièrement indépendant de l'évolutivité, même si en pratique ces deux aspects doivent naturellement être considérés ensemble.

13.1 Schémas XML ouverts

Les principaux mécanismes permettant de réaliser un schéma sous une forme ouverte sont les *jokers* ou *wildcards*. XML Schema connaît les les *Element Wildcards* (`xs:any`) et les *Attribute Wildcards* (`xs:anyAttribute`), et les deux sont des mécanismes servant à indiquer dans un Model Group ou dans une liste d'attributs que des éléments ou attributs supplémentaires et non spécifiés y sont autorisés. Ces jokers peuvent être régis de deux manières: d'abord, il est possible d'indiquer via `processContents` dans quelle mesure est exigée la validation des éléments ou attributs correspondant à une wildcard. Ensuite, on peut aussi indiquer, avec `namespace,` de quels espaces nominatifs les éléments ou attributs utilisés pour un joker doivent provenir.

L'autre manière de réaliser l'ouverture, plus subtile et souvent surprenante pour de nombreux utilisateurs ou programmeurs, est la *Type Substitution*, autorisée par défaut dans XML Schema. Cette substitution de types permet d'attribuer à un élément (elle ne peut être utilisée pour les attributs, ne serait-ce que pour des raisons purement syntactiques) un autre type dans un document. Sous une forme un peu différente sur le plan syntactique, la substitution de types intervient aussi dans les *Substitution Groups*. Avec l'attribut `blockDefault` du schéma, la substitution de types devrait d'abord être interdite de manière générale, puis être de nouveau autorisée uniquement aux endroits souhaités, qu'il faudra prendre en considération lors de l'implémentation. On utilise à cet effet `block` avec les Complex Types et les déclarations d'éléments.

La conception de schémas ouverts est, d'une part, très importante pour préparer la voie à des extensions ordonnées. Elle est, d'autre part, aussi fondamentale pour autoriser, déjà dans la première version des implémentations, l'ouverture aux endroits où des extensions sont à attendre pour plus tard. Ainsi par exemple, la première version de HTML définissait déjà que les navigateurs devaient ignorer les éléments et attributs inconnus, ce qui constituait, il est vrai, une forme plutôt sommaire d'ouverture, mais était nécessaire pour que le Web pût se développer avec la dynamique qui a fait son succès.

La conception de schémas ouverts doit poursuivre l'idéal qu'une implémentation de la première heure doit aussi pouvoir convenir pour des extensions développées et utilisées plus tard seulement. A cet effet, il suffira parfois d'ignorer de manière contrôlée les parties inconnues d'un document, mais il faudra peut-être aussi, notamment lors de l'utilisation de la substitution de types, faire en sorte que l'implémentation puisse travailler proprement au niveau des types, sans quoi les membres de groupes de substitution ne pourront pas être traités de manière adéquate

13.1.1 Recommandations

- **MAY**: Si un schéma ouvert est souhaité, de sorte que peuvent apparaître, dans des instances de celui-ci, des contenus qui n'y sont pas définis en détail, cela peut être réalisé au moyen de jokers (wildcards) pour les éléments et/ou les attributs. Ces jokers peuvent servir non seulement à conserver l'ouverture par rapport aux contenus inconnus, mais encore à réaliser l'ouverture envers les extensions prévues pour plus tard.

13.2 Schémas XML évolutifs

Après ces explications plutôt théoriques, nous voulons maintenant examiner comment assurer l'évolutivité d'un schéma XML. Pour cela, nous partons du principe qu'en cas d'extension d'un service Web, par exemple, on ne définira pas un schéma entièrement nouveau, mais que l'on utilisera l'ancien schéma pour procéder à une extension, par exemple en partant du mécanisme `xs:redefine`. Nous étudions plus en détail au chapitre 11 avec quelle exactitude on reliera les extension entre elles et dans quels cas on utilisera `xs:redefine`, `xs:include` ou `xs:import`. En première approche, nous supposons ici que le schéma est défini pour la première version dans un schéma XML et que les extensions seront ensuite des schémas autonomes qui importent la première version. Ainsi, et c'est ici le point essentiel, les espaces nominatifs sont conservés. L'exemple ci-dessous doit, pour la démonstration, utiliser des définitions globales à tous les endroits possibles:

```
<xs:schema targetNamespace=«http://example.com/person»
           xmlns:ns=«http://example.com/person»
           xmlns:xs=«http://www.w3.org/2001/XMLSchema»
           elementFormDefault=«qualified» blockDefault=«#all»>
     <xs:element name=«person» type=«ns:personType»/>
     <xs:complexType name=«personType»>
          <xs:group ref=«ns:personGroup»/>
     </xs:complexType>
```

```
      <xs:group name=«personGroup»>
            <xs:sequence>
                  <xs:element ref=«ns:name»/>
            </xs:sequence>
      </xs:group>
      <xs:complexType name=«nameType»>
            <xs:group ref=«ns:nameGroup»/>
      </xs:complexType>
      <xs:group name=«nameGroup»>
            <xs:sequence>
                  <xs:element ref=«ns:givenname»/>
                  <xs:element ref=«ns:surname»/>
            </xs:sequence>
      </xs:group>
      <xs:element name=«name» type=«ns:nameType»/>
      <xs:element name=«givenname» type=«xs:string»/>
      <xs:element name=«surname» type=«xs:string»/>
</xs:schema>
```

Maintenant, les extensions doivent être autorisées, c'est-à-dire que le schéma doit être réalisé de manière ouverte. Comme nous l'avons décrit ci-dessus, il existe plusieurs variantes pour cela. Des jokers peuvent être utilisés à différents endroits. Les déclarations suivantes, modifiées ou nouvelles, seraient adéquates afin d'ouvrir le nom pour d'autres attributs:

```
<xs:complexType name=«nameType»>
      <xs:group ref=«ns:nameGroup»/>
      <xs:attributeGroup ref=«ns:anyAttributeGroup»/>
</xs:complexType>
<xs:attributeGroup name=«anyAttributeGroup»>
      <xs:anyAttribute processContents=«lax» namespace=«##other»/>
</xs:attributeGroup>
```

L'élément `anyAttribute` a été défini comme Attribute Group autonome, afin de pouvoir être réutilisé, le cas échéant, à un autre endroit. Le schéma peut être ouvert pour d'autres éléments de manière très similaire:

```
<xs:complexType name=«nameType»>
  <xs:sequence>
    <xs:group ref=«ns:nameGroup»/>
    <xs:group ref=«ns:anyGroup»/>
  </xs:sequence>
</xs:complexType>
<xs:group name=«anyGroup»>
  <xs:any processContents=«lax» namespace=«##other»
          minOccurs=«0» maxOccurs=«unbounded»/>
</xs:group>
```

Ce genre d'évolutivité fonctionne bel et bien, mais uniquement parce que les espaces nominatifs autorisés de l'Element Wildcard ont été restreints avec `##other` à d'autres espaces nominatifs que le `targetNamespace`. Si des extensions avaient été autorisées dans le `targetNamespace` du schéma et qu'ensuite une extension ajoutant un élément optionnel

ait été effectuée dans le même espace nominatif, la règle de XML Schema interdisant les modèles de contenus non déterministes aurait été violée.

Si l'on veut renoncer aux jokers pour miser plutôt sur une évolutivité basée sur les types, on doit déclarer en conséquence les éléments et les types aux endroits où des types dérivés doivent être autorisés, car la valeur par défaut a été mise à `#all` avec `blockDefault` dans `xs:schema`. L'attribut `block` devrait ensuite être déclaré pour le type et pour l'élément, avec encore éventuellement la mention `substitution` pour l'élément si les groupes de substitution ne doivent pas être utilisés.

```
<xs:complexType name=«nameType» block=«restriction»>
  <xs:group ref=«ns:nameGroup»/>
</xs:complexType>
<xs:element name=«name» type=«ns:nameType»
            block=«substitution restriction»/>
```

De cette manière, les substitutions de types sont maintenant autorisées (mais seulement avec des types étendus, et non pas par des groupes de substitution). Que les éléments et leurs types doivent être redéclarés ensemble, comme dans ce cas, est laissé entièrement à la discipline du concepteur du schéma, XML Schema n'offrant aucune aide à ce propos.

13.2.1 Directives pour schémas évolutifs

Les exemples ont montré que l'on peut utiliser différents mécanismes de XML Schema pour ouvrir le schéma et le rendre ainsi extensible. Le choix du mécanisme que le concepteur va utiliser dépend naturellement, jusqu'à un certain point, de son goût personnel. On constatera toutefois que le mécanisme de substitution de types présuppose que l'on connaît et peut utiliser un type qui servira de base à l'extension, ce qui est peu réaliste notamment si l'on part d'une évolution du schéma plutôt distribuée et ne suivant pas une ligne droite. Le modèle le plus ouvert et le plus souple est l'extension par des éléments, que nous étudions ci-après. L'extension par une substitution de types est ainsi réservée aux cas dans lesquels il existe une coordination centrale des développements.

Dans notre exemple, l'Element Wildcard permet l'apparition d'éléments quelconques au point prévu pour les extensions. Cela est important, car l'un des objectifs est que tous les documents (donc notamment ceux qui comprennent des extensions) soient aussi validés par le schéma initial. Cela est nécessaire pour que l'évolutivité d'un système puisse être assurée sans restriction. Mais bien entendu, les documents doivent pouvoir non seulement être validés, mais aussi être traités. Nous nous retrouvons ainsi au point mentionné au début, à savoir que la sémantique peut être nécessaire. La règle par défaut pourrait être d'ignorer tout ce qui est inconnu, mais cela est très simpliste et peut parfois être indésirable, p. ex. si une certaine extension est essentielle et qu'il faille mentionner dans le document qu'elle ne peut pas être ignorée. Cette information peut être exprimée de différentes manières, dont nous en présenterons deux ici. Dans le premier cas, est défini pour le Document Element un attribut spécial qui contient tous les espaces nominatifs qui ne peuvent pas être ignorés:

```
<xs:element name=«person» type=«ns:personType»/>
```

```
<xs:complexType name=«personType»>
      <xs:group ref=«ns:personGroup»/>
      <xs:attribute name=«mustUnderstand»>
            <xs:simpleType>
                  <xs:list itemType=«xs:anyURI»/>
            </xs:simpleType>
      </xs:attribute>
</xs:complexType>
```

Ainsi, un programme peut constater de manière très simple si un document contient des extensions qu'il doit comprendre. Si le document contient des extensions inconnues, mais mentionnées dans l'attribut `mustUnderstand`, aucun traitement ne doit avoir lieu, mais l'application doit procéder d'une manière qui doit être prévue dans le modèle de traitement du système, p. ex. en produisant un cas d'erreur particulier ou en renégociant des formats échangés.

Une manière un peu plus élégante, mais demandant plus de travail au niveau du schéma, consiste à placer l'information dans les extensions elles-mêmes et à exiger p. ex. que chaque extension doit être dérivée d'un type spécial qui est prescrit dans le schéma initial et qui est p. ex. vide, ne contenant que les attributs qui sont nécessaires pour le traitement des extensions:

```
<xs:complexType name=«extensionType» abstract=«true»>
      <xs:attribute ref=«ns:mustUnderstand» use=«required»/>
      <xs:attribute ref=«ns:dependsOn»/>
</xs:complexType>
<xs:attribute name=«mustUnderstand» type=«xs:boolean»/>
<xs:attribute name=«dependsOn»>
      <xs:simpleType>
            <xs:list itemType=«xs:anyURI»/>
      </xs:simpleType>
</xs:attribute>
```

Ce type (qui est `abstract` et ne peut donc jamais apparaître comme instance directement dans un document) doit ensuite être pris comme base de toutes les extensions, de sorte que les deux attributs y sont définis. L'attribut `dependsOn` permet de définir, pour chaque extension, d'éventuelles dépendances envers d'autres extensions, de sorte qu'un peu plus d'informations puissent être exprimées sur la nécessité et les dépendances des extensions qu'avec le seul attribut `mustUnderstand`. Les attributs sont définis comme attributs globaux afin qu'ils doivent être utilisés avec leur espace nominatif dans les documents. Cela correspond mieux à l'attente des utilisateurs que si les attributs devaient y apparaître de manière non qualifiée.

Pour que cette solution fonctionne, les extensions doivent bien entendu respecter la convention prévoyant qu'elles doivent dériver de l'`extensionType`. Cette condition ne peut pas être formulée dans le schéma lui-même, car les jokers ne peuvent être restreints qu'à des espaces nominatifs, mais pas à des types. Etant donné cet inconvénient, on pourrait aussi envisager une approche basée sur la substitution de type, qui se présenterait de la manière suivante (l'`extensionType` qui vient d'être défini est aussi utilisé dans cet exemple):

```
<xs:complexType name=«nameType» block=«restriction»>
  <xs:sequence>
    <xs:group ref=«ns:nameGroup»/>
    <xs:element ref=«ns:extensionElement»
                minOccurs=«0» maxOccurs=«unbounded»/>
  </xs:sequence>
</xs:complexType>
<xs:element name=«extensionElement» type=«ns:extensionType»
            abstract=«true» block=«restriction»/>
```

Certes élégante, cette approche offre toutefois le gros inconvénient qu'elle n'aboutit pas à un schéma permettant de valider avec succès des extensions quelconques. Il s'agit donc ici d'un bon exemple d'un schéma évolutif, mais n'offrant pas l'ouverture souvent aussi souhaitée. C'est pourquoi il est conseillé d'utiliser plutôt des jokers que la substitution de types. Si l'on utilise les jokers, on doit garantir, par un modèle de traitement non définissable dans le schéma que, par exemple, une extension est toujours dérivée, comme décrit ci-dessus, d'une type prescrit.

D'une manière générale, il est conseillé de disposer, dans un environnement comprenant des schémas évolutifs et ouverts, d'un modèle de traitement bien défini, qui décrit non seulement les schémas et leurs extensions, mais aussi la manière de procéder avec les documents. Les attributs `mustUnderstand` et `dependsOn` que nous avons vus dans les exemples précédents vont aussi déjà dans ce sens, car ils comportent des informations devant produire un effet sur le traitement et doivent donc être pris en compte dans tous les cas.

13.2.2 Recommandations

Le choix d'un mécanisme parmi ceux que nous avons décrits ici et la manière de l'utiliser dépendent du modèle qui doit être mis en œuvre avec XML Schema ainsi que de la planification du développement possible ou voulu de ce modèle et du schéma y relatif. C'est pourquoi aucune recommandation ne peut être faite pour ce domaine.

N'existe malheureusement encore aucune méthodologie bien établie pour XML Schema en ce qui concerne l'évolution des schémas ou des méthodes bien définies d'adaptation d'un schéma aux nouveaux besoins. Il est donc indispensable de planifier sa propre stratégie afin d'avoir une ligne cohérente, au fil des différentes versions d'un schéma, sur la manière de procéder avec les nouvelles exigences.

14 Mécanismes complémentaires

Pour le proche avenir, XML Schema constitue le langage de schéma le plus important pour XML. Bien entendu, XML Schema comporte aussi des faiblesses et n'est pas en mesure d'implémenter dans un schéma certaines exigences très simples, par exemple:

- *Alternatives entre élément et attribut*: il serait souvent utile d'exprimer la possibilité d'utiliser soit un élément déterminé soit un attribut, mais pas les deux. XML Schema ne permet pas d'exprimer une telle relation entre éléments et attributs, du fait qu'il établit une séparation stricte entre les modèles de contenu des éléments et les listes d'attributs.
- *Relations entre les attributs*: alors qu'un langage d'expression très forte, qui permet de définir leur apparition, est disponible pour les éléments,les attributs sont mentionnés dans des listes sans que les relations entre eux puissent être définies, c'est-à-dire p. ex. sans que l'on puisse déclarer que deux attributs optionnels doivent toujours apparaître ensemble.
- *Relations entre les valeurs*: bien que les Simple Types permettent de définir le domaine des valeurs d'éléments ou d'attributs, aucune relation ne peut être définie entre des valeurs. On ne peut p. ex. pas définir qu'un attribut doit toujours avoir une valeur plus grande que celle d'un autre.

Les conditions de ce genre ou similaires, qui ne peuvent pas être représentées par XML Schema, devraient être documentées si elles sont nombreuses dans le modèle spécifique. Il est aussi possible d'utiliser d'autres langages de schéma, tels que RELAX NG [ISO19757-2] ou Schematron [ISO19757-3], qui complètent XML Schema. Ces langages de schéma complémentaires sortent du domaine d'application du présent document (qui se limite à XML Schema); leur utilisation devrait toutefois être envisagée, car elle permet, dans certaines circonstances, de mieux implémenter le modèle spécifique en XML.

15 Exclusion de responsabilité – Droits de tiers

Les normes élaborées par l'Association **eCH** et mises gratuitement à la disposition des utilisateurs, ainsi que les normes de tiers adoptées, ont seulement valeur de recommandations. L'Association **eCH** ne peut en aucun cas être tenue pour responsable des décisions ou mesures prises par un utilisateur sur la base des documents qu'elle met à disposition. L'utilisateur est tenu d'étudier attentivement les documents avant de les mettre en application et au besoin de procéder aux consultations appropriées. Les normes **eCH** ne remplacent en aucun cas les consultations techniques, organisationnelles ou juridiques appropriées dans un cas concret.

Les documents, méthodes, normes, procédés ou produits référencés dans les normes **eCH** peuvent le cas échéant être protégés par des dispositions légales sur les marques, les droits d'auteur ou les brevets. L'obtention des autorisations nécessaires auprès des personnes ou organisations détentrices des droits relève de la seule responsabilité de l'utilisateur.

Bien que l'Association **eCH** mette tout en œuvre pour assurer la qualité des normes qu'elle publie, elle ne peut fournir aucune assurance ou garantie quant à l'absence d'erreur, l'actualité, l'exhaustivité et l'exactitude des documents et informations mis à disposition. La teneur des normes **eCH** peut être modifiée à tout moment sans préavis.

Toute responsabilité relative à des dommages que l'utilisateur pourrait subir par suite de l'utilisation des normes **eCH** est exclue dans les limites des réglementations applicables.

16 Droits d'auteur

Tout auteur de normes **eCH** en conserve la propriété intellectuelle. Il s'engage toutefois à mettre gratuitement, et pour autant que ce soit possible, la propriété intellectuelle en question ou ses droits à une propriété intellectuelle de tiers à la disposition des groupes de spécialistes respectifs ainsi qu'à l'association **eCH**, pour une utilisation et un développement sans restriction dans le cadre des buts de l'association.

Les normes élaborées par les groupes de spécialistes peuvent, moyennant mention des auteurs **eCH** respectifs, être utilisées, développées et déployées gratuitement et sans restriction.

Les normes **eCH** sont complètement documentées et libres de toute restriction relevant du droit des brevets ou de droits de licence. La documentation correspondante peut être obtenue gratuitement.

Les présentes dispositions s'appliquent exclusivement aux normes élaborées par **eCH**, non aux normes ou produits de tiers auxquels il est fait référence dans les normes **eCH**. Les normes incluront les références appropriées aux droits de tiers.

Annexe A – Références et bibliographie

[eCH-0018] Erik Wilde, Hanspeter Salvisberg, Alexander Pina, *Meilleures pratiques XML*, eCH, Berne, Switzerland, eCH-0018, August 2005

[eCH-0033] Erik Wilde, *Description d'espaces nominatifs XML Namespaces*, eCH, Berne, Switzerland, eCH-0033, 2006.

[eCH-0050] Erik Wilde, *Hilfskomponenten zur Konstruktion von XML Schemas*, eCH, Berne, Switzerland, eCH-0050, 2006.

[ISO19757-2] International Organization for Standardization, *Information Technology — Document Schema Definition Languages (DSDL) — Part 2: Grammar-based Validation — RELAX NG*, ISO/IEC 19757-2, November 2003.

[ISO19757-3] International Organization for Standardization, *Information Technology — Document Schema Definition Languages (DSDL) — Part 3: Rule-based Validation — Schematron*, ISO/IEC 19757-3, June 2005.

[RFC2119] Scott O. Bradner, *Key Words for use in RFCs to Indicate Requirement Levels*, Internet RFC 2119, March 1997. http://www.ietf.org/rfc/rfc2119.txt

[RFC3066] Harald Tveit Alvestrand, *Tags for the Identification of Languages*, Internet RFC 3066, January 2001. http://www.ietf.org/rfc/rfc3066.txt

[RFC3986] Tim Berners-Lee, Roy Fielding, Larry Masinter, *Uniform Resource Identifier (URI): Generic Syntax*, Internet RFC 3986, January 2005. http://www.ietf.org/rfc/rfc3986.txt

[xml10third] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, François Yergeau, *Extensible Markup Language (XML) 1.0 (Third Edition)*, World Wide Web Consortium, Recommendation REC-xml-20040204, February 2004. http://www.w3.org/TR/2004/REC-xml-20040204

[xml11] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, François Yergeau, John Cowan, *Extensible Markup Language (XML) 1.1*, World Wide Web Consortium, Recommendation REC-xml11-20040204, February 2004. http://www.w3.org/TR/2004/REC-xml11-20040204

[xml11schema10] Henry S. Thompson, *Processing XML 1.1 Documents with XML Schema 1.0 Processors*, World Wide Web Consortium, Note NOTE-xml11schema10-20050511, May 2005. http://www.w3.org/TR/2005/NOTE-xml11schema10-20050511

[xmlns] Tim Bray, Dave Hollander, Andrew Layman, *Namespaces in XML*, World Wide Web Consortium, Recommendation REC-xml-names-19990114, January 1999. http://www.w3.org/TR/1999/REC-xml-names-19990114

[xmlns11] Tim Bray, Dave Hollander, Andrew Layman, Richard Tobin, *Namespaces in XML 1.1*, World Wide Web Consortium, Recommendation REC-xml-names11-20040204, February 2004. http://www.w3.org/TR/2004/REC-xml-names11-20040204

[xmlschema1sec] Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn, *XML Schema Part 1: Structures Second Edition*, World Wide Web Consortium, Recommendation REC-xmlschema-1-20041028, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028

[xmlschema2sec] Paul V. Biron, Ashok Malhotra, *XML Schema Part 2: Datatypes Second Edition*, World Wide Web Consortium, Recommendation REC-xmlschema-2-20041028, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028

Annexe B – Collaboration et contrôle

Hans-Ulrich Bucher, Avataris AG

Remo Dick, CSI DFJP

Claude Eisenhut, Eisenhut Informatik

Urs Gähler, VRSG

Jürg Hotz, canton de Thurgovie

Adrian K. Keller, SAG Software Systems AG

Willy Müller, USIC

Hubert Münst, Data Factory

Patrick Ostertag, Etat de Fribourg

Alexander Pina, Unisys (Suisse) SA

Fabian Probst, Haute école spécialisée de Soleure et de la Suisse du Nord-Ouest

Hanspeter Salvisberg, Unisys (Suisse) SA

Verena Sieber, T-Systems

Hans Ulrich Wiedmer, KOGIS - LT

Hansruedi Vock, OFIT

Erik Wilde, EPF Zurich

Annexe C – Abréviations et glossaire

Une liste des abréviations et un glossaire commenté, accompagnés de liens permettant d'approfondir le sujet, se trouvent sur l'internet à l'adresse http://dret.net/glossary/.

DOM	Document Object Model
DSDL	Document Schema Definition Languages
DTD	Document Type Definition
IANA	Internet Assigned Numbers Authority
IETF	Internet Engineering Task Force
RDBMS	Relational Database Management System
RFC	Request for Comments
URI	Universal Resource Identifier
XML	Extensible Markup Language
XSD	XML Schema Definition Language, häufig verwendete (aber nicht offizielle) Abkürzung für XML Schema