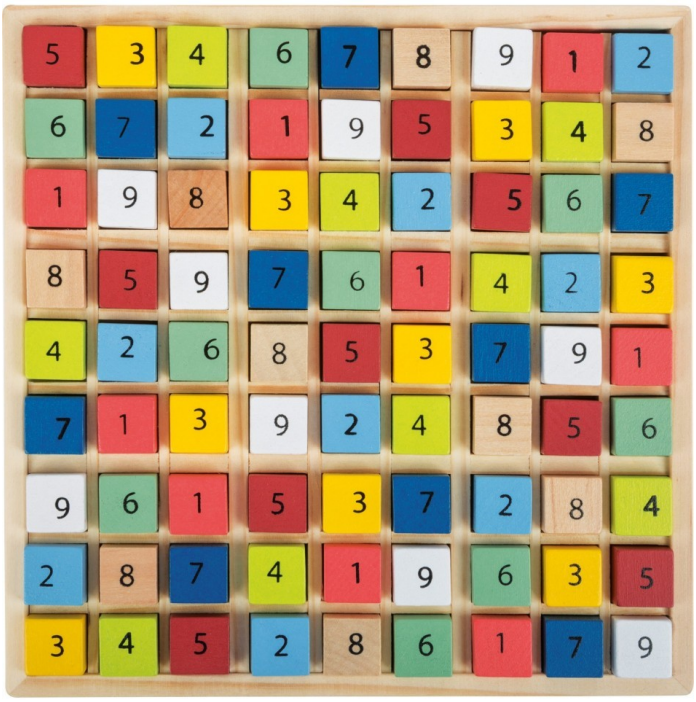
Projet SUDOKU

M2 MIASHS DCISS Université Grenoble Alpes



Préparé par :

Rosemond Marc Arold

Laurent Thévenon

Tran Eric

Professeur : Jérôme Gensel

Table des matières

I. Résumé

Ce projet avait pour objectif d'explorer les capacités du langage XML et des transformations XSLT à travers la modélisation et la validation d'une grille de Sudoku. Notre démarche a été guidée par une volonté constante d'optimisation, nous menant d'une structure riche en attributs vers une modélisation minimale et hautement algorithmique. Le projet s'est articulé autour de trois axes majeurs qui témoignent de cette progression technique.

Tout d'abord, l'évolution de la modélisation a été le point de départ de notre réflexion. Nous avons testé plusieurs structures XML, passant d'un modèle verbeux avec attributs de coordonnées à un modèle final sans attributs. Ce choix radical a permis de simplifier considérablement le schéma XSD et d'éliminer tout risque d'incohérence entre les métadonnées et la position réelle des éléments dans le document. Parallèlement, nous avons mené un travail de fond sur l'optimisation algorithmique en XSLT. Les premières versions utilisaient des boucles itératives dont la complexité était pénalisée par l'impossibilité d'interrompre le traitement. La version finale a résolu ces problématiques grâce à une approche récursive sophistiquée, permettant un arrêt prématuré dès la détection d'une erreur. Enfin, nous avons exploité la puissance de XSLT pour l'aide à la résolution via SVG. Au-delà de la simple validation, nous avons généré des interfaces graphiques dynamiques capables d'assister l'utilisateur en identifiant en temps réel les placements valides pour chaque chiffre. En conclusion, ce projet démontre que la simplicité de la structure de données, lorsqu'elle est couplée à une logique de traitement avancée, offre une solution élégante et performante pour la gestion de problèmes logiques complexes. Nous sommes aussi arrivés à la conclusion que toutes les modélisations dans le Sudoku ne se valent pas du point de vue de l’efficience algorithmique.

II. Schéma

Justification des choix de modélisation : avec ou sans attributs

Dans le cadre de ce projet, nous avons exploré deux approches pour la modélisation de la grille de Sudoku en XML : une modélisation utilisant des attributs pour définir les coordonnées de chaque cellule, et une modélisation simplifiée sans attributs.

Initialement, nous avons envisagé une structure où chaque élément `<cell>` portait des attributs tels que `*row*`, `*col*` et `*zone*` (par exemple : `<cell row="0" col="0" zone="0">5</cell>`). Cette approche semblait intuitive car elle permettait d'identifier explicitement la position de chaque valeur.

Problèmes rencontrés

Cependant, nous avons constaté que cette modélisation posait des difficultés majeures lors de la définition du schéma XSD. En effet, la validation par schéma XSD présente des limitations pour vérifier l'intégrité des données du Sudoku :

Contraintes d'unicité complexes : Bien que XSD permette d'utiliser `xs:unique` pour s'assurer que deux cellules n'ont pas les mêmes coordonnées, il est extrêmement complexe, voire impossible avec XSD 1.0, de vérifier les règles métier du Sudoku (absence de doublons de valeurs au sein d'une même ligne, colonne ou zone) en se basant sur ces attributs.

*Redondance et risque d'erreu*r : L'utilisation d'attributs introduit une redondance d'information. Si les attributs `*row*` et `*col*` ne correspondent pas à la position réelle de l'élément dans le fichier XML, cela peut entraîner des incohérences difficiles à gérer uniquement via le schéma.

XML avec des attributs

- ### Première version

Dans cette première approche, nous avons utilisé une structure plate où la racine `<sudoku>` contient directement les 81 cellules. Chaque cellule est identifiée par trois attributs obligatoires :

- `*row*` : l'index de la ligne (0-8).

- `*col*` : l'index de la colonne (0-8).

- `*zone*` : l'index de la région 3x3 (0-8).

Exemple de structure

```
<sudoku>
    <cell row="0" col="0" zone="0">5</cell>
    <cell row="0" col="1" zone="0">3</cell>
    ...
</sudoku>
```

Cette version est très verbeuse et nécessite de renseigner manuellement 243 attributs pour une grille complète, augmentant considérablement le risque d'erreur de saisie. Il a été facile de contourner ce problème en faisant un script Python.

- ## Deuxième version

La deuxième version introduit une hiérarchie pour mieux structurer les données. La racine contient un élément `<grille>`, qui lui-même contient des éléments `<row>`. Chaque ligne contient ses cellules respectives.

- L'élément `<row>` possède un attribut `*index*`.

- L'élément `<cell>` possède les attributs `*col*` et `*region*` (équivalent à la zone).

Exemple de structure :

```
<sudoku>
    <grille>
        <row index="1">
            <cell col="1" region="1">5</cell>
            <cell col="2" region="1"></cell>
            ...
        </row>
    </grille>
</sudoku>
```

Amélioration : La structure hiérarchique rend le fichier plus lisible pour un humain, mais la validation XSD reste limitée pour garantir que les attributs `*col*` et `*region*` sont cohérents avec la position réelle de la cellule dans l'arbre XML. De plus, il nous était impossible de vérifier si deux cellules étaient identifiées avec les même attributs.

XML sans attributs

Il s'agit de la version finale retenue pour le projet. Elle privilégie la simplicité en supprimant tout attribut de positionnement.

Exemple de structure :

```
<sudoku>
    <cell>5</cell>
    <cell>3</cell>
    <cell></cell>
    ...
</sudoku>
```

***Avantages** :*
- Légèreté : Le fichier est beaucoup plus compact.

- Fiabilité : Il n'y a aucun risque de contradiction entre un attribut de position et la position réelle de l'élément.

- Traitement : Les transformations XSLT peuvent facilement calculer la position d'une cellule en utilisant la fonction `*position()*` même si les calculs peuvent devenir plus complexe pour faire la feuille de style.

Nous avons hésité à y intégrer simplement des éléments `<row>` sans attribut, qui aurait contenu les cellules de chaque ligne. Si cela simplifiait l'identification de la ligne d'une cellule, cela rendait plus complexe l'identification de la colonne. De plus cela créait une sorte de hiérarchie entre les colonnes les lignes et les régions de chaque cellule, hiérarchie qui n'a pas de concrétisation dans les règles du sudoku.

Validation du schéma

Etant donné la simplicité de la structure que nous choisi pour représenter le sudoku, la mise en place de la validation du schéma a été assez aisée. Un élément racine sudoku est composé de 81 éléments `<cell>`, pouvant contenir une chaîne vide, ou un chiffre. Le nombre de cellules est garanti via *minOccurs* et *maxOccurs* (fixés à 81) et une expression régulière permet de valider le contenu des éléments `<cell>`

III. Feuilles de styles

Le développement des feuilles de style XSLT a suivi une courbe d'apprentissage technique, passant d'une approche procédurale simple à une architecture plus modulaire et optimisée. L'enjeu principal était de transformer une structure de données statique en une interface visuelle interactive tout en assurant la cohérence des règles du Sudoku.

Utilisation de variables pour conserver l'information

La première question qui s'est imposée à nous a été la suivante : une fois que l'on sait si une grille est complète ou correcte, comment garder ces informations pour les afficher. Comme les variables ne peuvent pas changer de valeur, nous avons donc décidé de les « remplir » grâce aux template. Ainsi, pour l'objectif 3 par exemple, une la variable erreur est définie dans le premier template (/sudoku) et englobe tout l'appel au template *while*. Ainsi, à la fin après l'appel à *while*, tout ce qui aura été écrit pendant l'exécution sera le contenu de la variable erreur, nous permettant d'afficher les informations utiles à la suite des tests.

```
<xsl:variable name="erreur">
<!-- Appel du template récursif -->
   <xsl:call-template name="while">
      <xsl:with-param name="pos" select='1'/>
      <xsl:with-param name="conflit" select="'non'" />
      <xsl:with-param name="vide" select="'non'" />
   </xsl:call-template>
</xsl:variable>
```

Pour l'objectif 4, nous avons amélioré cette méthode en mettant dans la variable chaque case ne pouvant être occupée par le chiffre testé.

```
<xsl:if test="($row1 = $row2) or ($col1 = $col2) or ($zone1 = $zone2)">
   <xsl:variable name="chaine" select="concat($row2,';',$col2)" />
    <xsl:value-of select="$chaine"/> and
</xsl:if>
```

Version schéma avec attributs

Première version

Dans cette première feuille adopte une structure que l'on pourrait qualifier de monolithique. L'essentiel de la logique est concentré dans un template unique ciblant l'élément `*sudoku*`. Ce choix s'explique par la volonté de traiter la grille comme un flux continu de données. Pour la validation, nous avons mis en place un mécanisme de détection de doublons qui illustre parfaitement les limites de l'approche itérative simple en XSLT 1.0. Le code suivant montre comment nous utilisons l'axe `*following-sibling*` pour comparer chaque cellule avec le reste de la grille. Cette version était adossée à la première version de structure, qui comprenait les attributs *@row*, *@col* et *@zone*

```
<xsl:for-each select="cell">
    <xsl:variable name="row1" select="@row" />
    <xsl:variable name="col1" select="@col" />
<xsl:variable name="col1" select="@zone" />
    <xsl:variable name="val1" select="." />
    <xsl:if test="not ($val1 = '')">
        <xsl:for-each select="following-sibling::node()">
            <xsl:if test="($val1 = .) and (($row1 = @row) or ($col1 =
@col) or ($col1=@col))">
                <!-- Détection du conflit -->
            </xsl:if>
        </xsl:for-each>
    </xsl:if>
</xsl:for-each>
```

Cette structure de boucles imbriquées est le cœur du problème de complexité que nous avons soulevé. Pour chaque cellule (81 au total) nous parcourons toutes les cellules suivantes. Mathématiquement cela nous place dans une complexité de O(n(n+1) / 2), soit plus de 3000 passages. Le rendu graphique, quant à lui, est généré par une itération similaire où chaque cellule est dessinée via un élément `<rect>` dont les coordonnées sont calculées par une opération arithmétique directe sur les attributs : `x="{@col * 50}" y="{@row * 50}"`. Cette méthode, bien qu'efficace pour l'affichage, ne permet aucune abstraction graphique.

Deuxième version

Dans la deuxième version avec les attributs, on introduit une rupture technologique en utilisant des fonctionnalités avancées de XSLT pour pallier les manques de la première version. La première grande différence réside dans la définition de clés d'indexation (`*xsl:key*`), qui permettent de transformer la recherche de doublons en une opération de consultation de table de hachage. Voici comment nous avons structuré ces indexes :

```
<xsl:key name="cols" match="cell" use="concat(@col, '-', text())"/>
<xsl:key name="regions" match="cell" use="concat(@region, '-',
text())"/>
<xsl:key name="rows" match="cell" use="concat(../@index, '-',
text())"/>
```

L'utilisation de `*concat()*` dans l'attribut `*use*` est un choix délibéré pour créer une signature unique combinant la position et la valeur. Cela permet au processeur XSLT de regrouper instantanément toutes les cellules ayant la même valeur dans une même colonne ou région. La validation s'en trouve simplifiée et devient beaucoup plus lisible dans le template `*verifier-statut*` :

```
<xsl:if test="count(key('cols', concat(@col, text()))) > 1 or
              count(key('rows', concat(../@index, text()))) > 1 or
              count(key('regions', concat(@region, text()))) > 1">
    <xsl:text>1</xsl:text>
</xsl:if>
```

Sur le plan de la structure, nous avons abandonné le `*for-each*` massif au profit d'une délégation par templates (`*xsl:apply-templates*`). Le template racine se contente de dessiner le cadre et les lignes épaisses du Sudoku, tandis que le template `*match="cell*"` s'occupe du rendu individuel. Ce choix de conception permet une séparation nette des responsabilités : le contenant (la grille) ne se préoccupe pas du contenu (la cellule). Cependant, malgré cette élégance architecturale et l'optimisation des recherches via les clés, le problème de la complexité globale persiste.
L'algorithme doit toujours itérer sur les 81 cellules pour déclencher ces vérifications, en contrôlant l’ensemble des autres cellules, soit une complexité de $O(n^2)$. C'est cette frustration technique qui a motivé notre passage final vers une logique récursive.

Version finale sans attributs

La feuille de style `objectif3xsl` représente l'aboutissement technique de notre projet. En supprimant les attributs de positionnement, nous avons été contraints de réinventer la logique de parcours et de validation en nous appuyant sur des concepts mathématiques et une structure récursive avancée.

Le template racine et l'orchestration récursive

Le template racine `*match="/sudoku"*` ne se contente plus d'initialiser le document ; il lance le moteur de vérification via un appel au template nommé `*while*`. Contrairement aux versions précédentes qui utilisaient des boucles `*for-each*`, nous avons ici fait le choix de la récursivité. Ce template `*while*` agit comme une boucle de contrôle qui parcourt les cellules une par une, de la position 1 à 81.

L'avantage crucial de cette approche est la gestion des paramètres `*conflit*` et `*vide*`. À chaque étape, le template vérifie si une erreur et une case vide ont déjà été détectées. Si c'est le cas, la récursion s'arrête immédiatement grâce à une condition de test :

```
<xsl:if test="($pos &lt; 82) and not (contains($conflit, 'oui')) and not
(contains($vide,'oui'))">
    <!-- Traitement de la cellule suivante →
</xsl:if>
```

Ce mécanisme de coupe-circuit résout enfin le problème d'inefficacité des versions précédentes. Le processeur n'effectue plus de travail inutile : dès qu'il y a à la fois une erreur et une case vide, le diagnostic est rendu, économisant ainsi potentiellement des milliers d'opérations sur une grille invalide et incomplète. Dans le pire des cas (aucune erreur et aucune case vide) le nombre de tests sera de n(n+1) * 2. Notons qu'après avoir rencontré des difficultés à cause du formatage automatique du code, nous avons remplacé *$conflit='oui'* par *contains($conflit, 'oui')* pour éviter les erreurs.

La conversion mathématique des positions

L'absence d'attributs `*@row*` et `*@col*` nous oblige à calculer la position logique de chaque cellule à partir de sa position dans l'élément soduku (1 à 81). Nous avons utilisé trois formules mathématiques fondamentales pour reconstruire la géométrie du Sudoku :

Calcul de la Ligne: floor(($pos - 1) div 9) + 1

Cette formule utilise la division entière. En soustrayant 1 à la position, puis en divisant par 9 (le nombre de colonnes), le résultat `*floor*` nous donne l'index de la ligne. On ajoute 1 à la fin pour revenir à un indexage que nous voulons (1 à 9).

Calcul de la Colonne: ($pos - 1) mod 9 + 1

Ici, l'opérateur modulo (`*mod*`) récupère le reste de la division par 9. Cela permet de boucler de 1 à 9 de manière répétitive pour chaque ligne.

Calcul de la Région (Zone 3x3) : `(floor(($row - 1) div 3) * 3 + 1 + floor(($col - 1) div 3))`

C'est la formule la plus complexe. Elle décompose la grille en blocs de 3x3. La partie `*floor(($row - 1) div 3) * 3*` détermine le bloc vertical (0, 3 ou 6), tandis que `*floor(($col - 1) div 3)*` détermine le décalage horizontal (0, 1 ou 2). Le `+ *1*` final harmonise l'indexation. Cette formule permet d'identifier l'appartenance à une région sans aucune information structurelle préalable dans le XML.

La logique de vérification des doublons (`siblings`)

Une fois les coordonnées calculées, la validation proprement dite est déléguée au template `*siblings*`. Ce template est lui aussi récursif. Pour une cellule donnée à la position `*pos*`, il va comparer sa valeur avec toutes les cellules suivantes (`*pos2*`).

```
<xsl:if test="($row1 = $row2) or ($col1 = $col2) or ($zone1 = $zone2)">
    conflit
</xsl:if>
```

Ce template intègre également une condition d'arrêt : si un conflit est détecté entre deux cellules, le mot-clé "conflit" est retourné, et la récursion s'arrête. Cette imbrication de deux niveaux de récursion (le parcours global et le parcours des voisins) permet d'obtenir une validation extrêmement rigoureuse et performante, tout en conservant un fichier XML d'une grande simplicité.

Coloration

Une problématique complémentaire, objectif 4, de notre projet consistait à aider l'utilisateur dans la résolution du Sudoku en affichant graphiquement toutes les cases où un chiffre spécifique (de 1 à 9) peut être légalement placé, compte tenu de la configuration actuelle de la grille.
Cette fois, il n'était pas possible d'utiliser de limiter le nombre de tests, car toutes les cases doivent être explorée. Notre feuille de style parcours donc toutes les cellules, et si une cellule contient le chiffre recherché, toutes les autres cellule seront parcourues. Si dans ce parcourt une cellule est vide, alors une vérification est faite pour voir si il autorisé d'y insérer le chiffre recherché. Si non, elle sera marqué et ses coordonnées seront inscrites dans la variable `*blocked_tiles*

Le rendu final utilise ensuite cette variable pour adapter l'affichage via une structure conditionnelle `*xsl:choose*`. Pour chaque cellule, le processeur vérifie si ses coordonnées figurent dans la liste des cases bloquées afin de définir le style du rectangle SVG :

```
<xsl:choose>
    <xsl:when test="contains($blocked_tiles, concat($row,';',
$col))">
        <rect width="50" height="50" x="{$colSVG+10}"
y="{$rowSVG+10}"
              style="fill:rgba(235, 10, 10, 1);stroke-
width:3;stroke:black" />
    </xsl:when>
    <xsl:otherwise>
        <rect width="50" height="50" x="{$colSVG+10}"
y="{$rowSVG+10}"
              style="fill:rgb(255,255,255);stroke-
width:3;stroke:black" />
    </xsl:otherwise>
</xsl:choose>
```

Cette logique de coloration permet de marquer instantanément en rouge (`*rgba(235, 10, 10, 1)*`) les zones d'influence des chiffres déjà placés. Parallèlement, une seconde vérification est effectuée pour afficher le chiffre suggéré en vert uniquement dans les cases qui ne sont ni occupées, ni bloquées :

```
<xsl:choose>
    <xsl:when test="(.='') and not (contains($blocked_tiles,
concat($row,';',$col)))">
        <text x="{$colSVG +26}" y="{$rowSVG +45}" font-size="2em"
fill="green">
            <xsl:value-of select="$number" />
        </text>
    </xsl:when>
</xsl:choose>
```

Cette approche transforme le document de données en un véritable outil d'aide à la décision, offrant une vue analytique claire qui facilite l'identification des coups possibles par simple lecture graphique. L'utilisation de couleurs contrastées et de conditions logiques strictes assure que l'utilisateur reçoit une information fiable et immédiatement exploitable pour résoudre sa grille.
Par ailleurs, la logique étant exactement la même quelque soit le chiffre recherché, nous avons simplement déclaré une variable *number* à la racine de la feuille de style. Nous n’avons ensuite eu qu’à changer sa valeur pour créer une feuille de style par chiffre.

Conclusion

En travaillant sur ce projet, nous avons pu prendre conscience à la fois de la puissance des technologies XML, mais aussi des contraintes qu'elles imposent.

Nous étions partis sur une version instinctive de la représentation du sudoku en utilisant des attributs colonne, ligne et région pour chaque région, avant de constater qu'il était impossible de garantir que le document était correct.

Bien étant à arrivé à une version fonctionnelle de la vérification, nous avons choisi de pousser la réflexion plus loin en limitant au maximum le nombre de tests effectués. Bien que cela nous ait rendu la tâche bien plus difficile, ce choix a surtout été l'occasion d'explorer plus en profondeur les possibilité offertes par XSLT, en particulier en simulant des structures du type « tant que » qui ne sont pas directement intégrées au langage.

Enfin, nous avons pu constater comment bien utilisé, XSLT et SVG peuvent s'imbriquer pour créer des représentations à partir de fichier XML plus « classiques ».