Antenna House Formatter と CSS を使ったルーズリーフ出版

Balisage: Markup 会議 2019
2019 年 7 月 30 日～8 月 2 日

課題

ルーズリーフ出版とは、以前に印刷された文書のページ番号は変更せずに、内容の更新を行う出版方法です。文書の更新により新しいページが作成されると、それらのページには元のページ番号に修飾子を加えたページ番号、例えば「10.1」、「10.2」などが付与されます。このようなページは「ポイントページ」と呼ばれます。文書の更新は新しく追加された、もしくは変更のあったページのみ作成され、対象の文書に手作業で挿入することにより、マスター文書の最新版となります。

低コストのプリンターとデジタルドキュメントの配信が登場する以前は、ルーズリーフ出版が一般的でした。PDF の再生成や、レーザープリンターで数千ページを数分で印刷できる今日では、ルーズリーフ出版の必要性はほとんどなくなりました。

まだ需要が存在する分野の 1 つは、法的な出版を行う地方自治体の条例分野です。

法的文書、特に成文化された地方自治体の条例および規制には、いくつかの実用上の問題があります。

- 膨大な量の文書になる傾向があります。市の条例の場合、2000 ページ程度が一般的です。
- 閲覧者や関連文書により、古い版のページ番号が参照されることがあります。
- 条例は頻繁に改正されます:市町村議会ではたびたび条例の新設や改正が行われ、成文化された条例が変えられる可能性があります。
- 文書のライフサイクルは非常に長いです。自治体は条例を定期的に「リフロー」してページ番号を刷新して全体を再公開できますが、リフローは 10 年に 1 度もしくはそれ以下の頻度でしか行えません。
- 自治体職員や条例を取り扱う職業の人は、日常業務のために印刷された条例を持っていますが、条例に変更があるたびにこれらの印刷物をすべて更新することは大きなムダです。

法的に重要な意味を持つ 2000 ページもの文書のレビューと品質保証の問題は別としても、出版物が膨大なサイズであり、頻繁に更新があり、また印刷物が日常的に使用されていることから、更新があるごとに条例全体を再印刷することは困難です。

原則として、自治体自体は成文化を行いません。地方自治体条例の成文化と公開はサービスであり、そのようなサービスプロバイダーの 1 つが、米国の地方自治体条例の最大のサプライヤの 1 つである Municode です。

Municode は長年にわたって Xyvision Parlance Publisher（XPP）製品を使用して、地方自治体条例用のルーズリーフページを作成していました。XPP で作成はできましたが、SGML や XML 出版システムとしてではなく、従来型の組版システムとして使用していました。編集者は初期の体系化されたマークアップとコンピューター化された組版である、XPP の組版形式（「gencode」）で直接編集していました。

Municode は、XPP システムを最新の XML ベースの出版パイプラインに変更する必要があることに気付きました。HTML5 ベースのボキャブラリをソースとして開発し、レイアウト技術として CSS ページ組版を使用し、組版エンジンとして Antenna House Formatter（AHF）を選択しました。Antenna House Formatter は CSS ページ組版を実装しています。

ただし、AHF 自体にはルーズリーフ出版機能がないため、ルーズリーフ処理を実装する必要があります。特に、AHF（および CSS）には、ポイントページ番号またはそれらへの参照を生成する直接的な機能がありません。

この著者は以前、フィールドエージェントが使用する出版物の更新を公開するという米国連邦機関への提案に関連して、Antenna House エリアツリーの後処理を使用して変更ページを生成するアプローチを設計していました。提案は受け入れられませんでしたが、提案開発プロセスの一環として Antenna House にその設計書を提供しました。Antenna House は私を Municode に紹介し、私はルーズリーフ出版を実装するため Municode に雇われました。

Municode は、ページ組版のスタイル設定に XSL-FO ではなく CSS を使用することを要求していたため、私は、CSS ページ組版処理に必要な XML 入力を生成するために、ページ組版用のスタイルと前処理も開発しました。

Antenna House Formatter がページ組版エンジンとして選択されたのは、当時、唯一 Municode のすべての組版要件を満たした CSS ページ組版を実装していたためです。もし Municode が XSL-FO を要求していたら、ポイントページ番号を生成するために必要な後処理を可能にするため、必要なレイアウト機能を提供しエリアツリーも生成可能な Apache FOP を使用していたかもしれません。

ルーズリーフ出版の課題

ルーズリーフ出版の課題の 1 つは、文書の 2 つのバージョン間でどのページが変更されたかを判断することです。Municode はこれを編集作業において手動で行うため、ページ組版の違いを自動的に判断することは、最初の実装の要件ではありませんでした。変更の自動判断は、将来的な

課題です。

主な課題は、入力として XML ソースを受け取り、出力として「変更パッケージ」を生成する自動プロセスを実装することでした。

ハイレベルのパイプライン処理は次のとおりです。

1. 編集者は、変更されたページの開始と終了のマークを含む XML ソースを準備します。開始は常に、以前のバージョンのページの開始に対応し、終了は変更が終了する場所です。
2. 入力 XML ソースは XHTML を生成するために前処理し、必要に応じて一般的には CSS ページ組版を可能にし、また具体的には、変更ページの生成を可能にするために拡張されます。
3. 拡張された XHTML は、CSS スタイルを使用して AHF によってレンダリングされ、最初のエリアツリーが生成されます。
4. 最初のエリアツリーが処理され、ポイントページ番号とそれらのページを参照するページのページ番号が更新されます。変更パッケージが作成されている場合、変更されていないページはすべて除外され、生成された「更新の説明」の節、目次、表紙など、変更されたページとパッケージに必要な他のページのみを反映するエリアツリーが作成されます。
5. マスターページの履歴データベースが更新され、更新されたバージョンの文書のページの詳細が反映されます。これには開始ページと終了ページの ID を持つ要素からのマッピングが含まれます。
6. 更新されたエリアツリーは AHF によって PDF にレンダリングされます。

ポイントページを作成する際の主な処理の課題は、変更された一連のページ内のどの場所でポイントページが必要かを知ることです。

入力ソースは変更されたページの開始をマークし、以前のバージョンから変更のあった最初のページ、および必要に応じて変更されたページに *続く* ページのページ番号を指定します。

オーサリング時には作成者は変更によって生成されるページ数が分らないため、最初のポイントページがどこから始まるか、またはポイントページが必要かどうか分かりません。変更が節の最後にあり、次の節が新しいページ番号シーケンスを開始する場合、変更に続くページ番号を指定する必要はありません。それ以外の場合、変更後に続くページ番号を指定する必要があります。

変更の開始と変更後に続くページのページ番号が振られ、ページが設計されると、ページの数が同じソースの元のページ数より大きいかどうかを判断できます。そのため、変更されたどのページにポイントページ番号が必要かを判断できます。その後、それらのページ番号を更新し、それらのページで始まる要素を参照しているページ番号を更新できます（たとえば目次エントリ、相互参照など）。

別の課題は、変更パッケージの目次を生成することです。

変更パッケージを作成するときは、文書全体の目次を複製する必要があります。ただし、レンダリングされた文書内の唯一の信頼できるページ番号は、変更されたページ自体のページ番号です。他のすべてのページ番号はさまざまな理由で信頼できません。特に、以前のバージョンが完全に異なるツール（AHF ではなく XPP）で作成された可能性、AHF 自体のバージョン間でレイアウトの詳細に単純に違いがある可能性、またはページ組版の詳細に影響する CSS の変更がある可能性があります。

したがって、一連の変更に含まれなかったページにある要素の目次に反映されるページ番号には、それらの要素に最後に公開されたバージョンで振られたページ番号を反映する必要があります。これには、公開時に特定のバージョンで公開されたページ番号に要素を関連付ける、何らかの形式のページ組版データベースが必要です。

最後の課題は、変更パッケージの「更新の説明」と「有効ページのリスト」の節を生成することです。

更新の説明では、現在の更新パッケージ内で変更があったページのセットごとに、以前のバージョンから削除されるページと挿入される新しいページを指定します。このためには、そのページ番号が前のバージョンと現在のバージョンで何であるかを知る必要があります。

有効ページのリストは、出版物の各ページについて、最新の更新で何が（どのバージョンで）更新されたかを示します。これには更新履歴を取得する各物理ページの履歴が必要です。

これら 2 つの履歴データセットを使用すると、更新の説明と有効ページのリストを完全に生成できます。これらの加工物は両方とも、従来のシステムでは手作業で作成する必要がありました。タスクは面倒で間違いが起きやすく、生成プロセスにかなりの時間を要し、Municode にとってタイムリーな更新の障害となっていました。

最後に、Municode は自治体の条例を Web でも公開しているため、もう 1 つのビジネス要件は、印刷物製作と Web 配信用の互いの生成プロセスに手作業で介入することなく、両方を直接提供できる単一のオーサリングソースを持つことです。

従来のシステムでは、XPP 組版ソースはまず中間 XML 形式に変換され、そこから最終的な配信 HTML が生成されました。このプロセスは 100%自動ではなく、常に 100%正しいオンライン結果が得られるわけでもなく、さらに時間もかかりました。

CSS ページ組版の課題

CSS ページ組版[1]にはいくつもの課題があります。

- CSS ページ組版の仕様は、編集面または機能面で完全ではありません

- CSS ページ組版の仕様は、多数の個別の仕様に分散しています。
- CSS 自体は HTML のスタイリングに最適化されています

純粋に標準で定義されているレイアウトと印刷機能の点で XSL-FO と比較すると、CSS にはヘッダとフッタの完全な制御、任意の要素または属性にリンクを与える方法がない、XSL-FO のテーブルマーカー機能、およびその他の XSL-FO のより難解であまり使用されない機能、特にアジア言語のサポートに関する機能など、いくつかの重要な機能が欠けています。

ページ組版に関する CSS の明確な利点は、スタイル自体を簡単に指定できることです。CSS は、CSS スタイルシートがスタイル設定されているソースから完全に分離されているという理由で、客観的に見て XSL-FO よりもスタイルを指定するのがはるかに簡単です。一方、XSL-FO は変換によって生成される必要があるソース形式です。XSL-FO を生成するには、データ処理側とスタイリング側が統合されている影響により、2 つの関連を分離するのが困難であるという問題があります。

ページ組版に CSS を使用するもう 1 つの利点は、必要に応じてブラウザと印刷用に同じコアスタイルを使用できることです。

CSS はそれ自体が変換言語ではないため、データ処理とスタイリングの関連を分けなければいけません。

実践的な観点からは XSL-FO を知っているか、学びたい人を見つけるよりも、CSS を知っているか、学びたいと思っている人を見つける方がはるかに簡単です。たとえ 2 つの技術が機能的に同等であったとしても、CSS を使える人を見つけやすいということは大きな利点となります。[2]

スタイル言語として、CSS は装飾のみを行うことができ、複雑な要素の並べ替えや作成はできません。これにより CSS のアーキテクチャと構文がシンプルになり、レンダリング速度が最優先であるブラウザで使用するための要件となりますが、たとえ HTML でオーサリングした場合でも、典型的なソースドキュメントをオーサリングしたとおりに完全にスタイル設定することはできません。

CSS のもう 1 つの重要な制約はセレクタです。CSS セレクタは現在の要素の先方を見ることができません。つまり、子孫または後続要素のプロパティに基づいて直接要素のスタイル設定をすることはできません。

さらに、CSS は HTML のスタイリング専用に設計されているため、拡張機能なしで任意の XML を完全にスタイル設定することができません。

したがって CSS をページ組版に使用するには、レンダリングに適した HTML を効果的に生成する必要があります。同じ拡張タスクを XML に適用することはできますが、CSS は HTML 用に最適化されているため、単純に HTML を生成する方が簡単で、より適切です。ほとんどではないにしても、多くの XML ボキャブラリには CSS ページ組版対応の HTML を生成するために再利用できる HTML 生成変換が既に存在します。これにより完全に別個の XSL-FO 生成変換も必要なくなり、

大幅な節約になります。

この生成ステップは、ブラウザで JavaScript によって実行されることが多い生成に似ています。どちらの場合も、生成環境のレンダリング要件を満たすために、ソース HTML を拡張または変更する必要があります。

CSS ページ組版を有効にするために完了しなければいけない事項としては、

- 目次、巻末索引、および類似のナビゲーション構造の生成。
- 構造化されたヘッダとフッタを作成するために使用される要素の生成。たとえば書式設定が異なる複数行のヘッダ、または HTML の個別の要素を必要とするインライン書式設定などです。
- @class 値またはその他の考えられる手がかりを追加して、CSS スタイリングを可能に（先読みして）またはより便利にする。
- ソースの順序に関係なく表示される要素の並べ替え。たとえば、図のキャプション要素を図の上部から図の下部に移動したり、メタデータ要素または属性を使用して表示されるコンテンツ（著作権ページや各記事または章の著作者など）を合成する。
- ラッパー構造を追加して特定のフォーマット効果を有効にするか、スタイリングを簡単にする。
- 作成されたさまざまなマークアップパターンを持つ要素のマークアップを標準化する。たとえば、リスト項目に段落要素を追加して CSS スタイルシートを単純化する。
- CSS だけでは生成が困難または不可能なテキストを生成する。

作成されたソース自体が XHTML である場合、この変換は比較的単純です。ソースが他のボキャブラリである場合、CSS ページ組版対応の HTML 生成に適応する既存の HTML 生成変換がある可能性があります。

ページ組版対応の HTML を生成する必要性に加えて、CSS ページ組版の仕様には、より複雑なレイアウトに必要な機能が欠けています。したがって CSS ページ組版の完全な実装には、この機能のギャップを埋めるための独自の拡張機能が必要です。AHF は、基本的にすべての XSL-FO 機能または AHF 拡張を、対応する CSS 拡張にマッピングすることによりこれを行います。たとえば、AHF は追加のページシーケンス型を提供して、「last-」と「only-」のページシーケンスの作成を可能にします。

最後の課題は、ベース CSS 構文にパラメータ化がないことです。「less」などのツールは、CSS スタイルシートをパラメータ化およびモジュール化する方法を提供しますが、プロジェクトの開始時に適切な Java ベースの less コンパイラは見つかりませんでした。したがって、CSS スタイルシートには、特にページマスタールールで多くの重複がありますが、このコードは開発後に劇的な変化

をしなかったため、パラメータ化の欠如は小さな問題であることが判明し、その解決は優先事項とはなりませんでした。

CSS スタイルシートを実装する際の主な課題は次のとおりです。

- 特定のレイアウト機能に関連する定義を、関連する W3C 仕様の中で見つける。
- AHF が仕様で定義された特定の機能を実装しているかどうかを判断する。
- 複雑なレイアウト要件においては、AHF を使用した最適なソリューションを判断する。
- 改ページを動的にコントロールする。

ほとんどのレイアウト要件では、CSS の開発は通常の CSS 技術の範疇の単純な応用で可能でした。

複雑な要件としては以下のものがあります。

- ページの first または last 値を反映する必要がある柱（ランニングヘッダとランニングフッタ）の要素の境界を越えたカウンターと変数の管理。
- 改ページの管理。改ページ制御の CSS セマンティクスは、XSL-FO ほど明確ではない。特に、CSS には「keep together always」または「keep with next always」コントロールがない。CSS の Keep は、本当に「ヒント（hints）」です。これにより、ページの下部にある節見出しと、コンテンツが介在しないサブセクションの見出しの間など、残念な改ページが生じることがあった。改ページをより適切にコントロールするには、AHF 拡張機能を使用する必要があった。
- 幅の広いページ端領域のサイズとレイアウトの制御。ページ端領域の CSS 設計では、単一の領域が端領域のほとんどまたはすべてを占めることを明確に許可していない。これにより、長いコンテンツ（たとえば、長い節見出し）を持つ右揃えまたは左揃えのヘッダを作成することが困難になっている。

CSS ページ組版用に XML を準備する

CSS は変換言語というよりは宣言型スタイル言語です。これにより CSS が比較的単純になり、視覚と動作スタイルの定義とソース XML に適用されるデータ処理との間の関連が明確に分離されますが、ほとんどの XML は完全に作成した通りにスタイル設定できません。したがって作成された XML は、CSS を使用して完全にスタイル設定できる形式に変換する必要があります。さらに、CSS は HTML のスタイル設定に最適化されているため、任意の XML のスタイル設定に必要な機能が欠けています。たとえば、CSS は、すべてのリンクが@href と URL をアドレス指定に使用する HTML <a> 要素によって表されることを想定しており、リンク動作を任意の要素または属性に

関連付ける方法を提供していません。

CSS を任意の XML に直接適用して妥当な結果を得ることは技術的には可能ですが、重要なページレイアウトの場合、その結果は必然的に不完全になります。HTML であっても、作成者が、柱（ランニングヘッダやフッタ）の様に自動的に生成またはレンダリングされるものではなく、コンテンツのみを担当し、HTML が通常の方法で作成されている場合、それでもソース HTML はページレイアウトの要件を満たすためにある程度の拡張を必要とします。

したがって、ソース XML が何であっても（たとえソースが HTML であっても）ページ組版対応の XML を作成するには、ある程度の変換が常に必要です。

この変換を XSL-FO の使用に必要な変換と比較します。XSL-FO の場合、対象となるボキャブラリは常に XSL-FO であり、スタイルの詳細は生成された XSL-FO マークアップおよびコンテンツに埋め込まれます。CSS ページ組版の場合、対象となるボキャブラリは任意の XML にすることができますが、ほとんどの場合 HTML であり、変換はソース XML の構造および意味上（セマンティック）の詳細を変更するだけで、スタイルの詳細はすべて CSS スタイルシートに残ります。

オーサリングしたソースが HTML である場合、変換は実際には「拡張」タスクであり、必要に応じて追加や並べ替えを行いますが、それ以外は HTML がオーサリングされたとおりに保持する ID 変換のみです。

オーサリングしたソースが HTML でない場合、必要なすべての拡張が適用される HTML への変換が最も効果的です。ソース XML ボキャブラリに HTML 変換がすでに存在する場合、必要なのは、CSS ページ組版要件を満たすために必要な拡張を追加することだけです。

フル機能のページレイアウトを持つ技術文書、法律文書、一般書籍、およびその他の高度なデザインの出版物など 100%自動化された構成が適している文書においては、以下のレイアウト要件で、作成されたまたは既存の HTML 変換によって生成された HTML の拡張が必要です。

- 複数行のヘッダやフッタ、または印字の違いがあるコンテンツなどの構造化されたページ端領域コンテンツ（つまり、柱（ランニングヘッダやフッタ）に反映されるタイトルの太字または斜体の単語）。
- スタイルを決定するために子孫または後続の要素のプロパティに依存する要素。
- 作成された形式とは異なる順序または構造で提示されるコンテンツ。たとえば、図のタイトルを図コンテナの先頭から下に移動したり、メインフローにメタデータ要素を反映したりする。
- CSS を使用して簡単には生成できない、またはまったく生成できないテキスト（たとえば、計算またはソースの文字列処理を必要とするテキスト）

さらに属性、または厳密に必須ではないがスタイル定義を容易にする要素を追加することにより、CSS スタイル定義の作成および保守を容易にすることができます。たとえば、必要なセレクタの単

純化や、変数マークアップパターンを単一の一貫したパターンに標準化することで可能です。

構造化したページ端領域コンテンツの提供

CSS は、スタイル宣言を介してソースのコンテンツをさまざまな場所に反映するための 2 つのメカニズムをもっています。

- 文字列変数
- 要素変数

文字列変数は、コンテンツまたは属性値を名前付き変数にコピーし、content:プロパティで使用できるようにします。

```
chapterTitle {
    string-set: chapterTitle content();
    display: none;
}
```

string-set：プロパティは、<chapterTitle>要素のテキストコンテンツを「chapterTitle」という名前の変数として取得します。変数は content:で使用できます。

```
@page portrait:right {

  @top-center {
    content: string(chapterTitle, last);
    vertical-align: bottom;
    margin-top: 0.25in;
    margin-bottom: 2pc;
    text-transform: uppercase;
  }

  ...
}
```

ここでは、「chapterTitle」変数に設定された last の値が、このタイプの上部中央ページ端領域のコンテンツとして使用されます。

要素変数は、ソースフローから要素を削除し、ページ端領域で使用できる変数として取得します。要素は、ソース内で発生する時点で有効なスタイルを使用してスタイル設定されます。

```
sectionTitlesMultiline {
```

```
    position: running(runningHead);
  }
```

ここで、position：プロパティは、<sectionTitlesMultiline>要素の「position」を「runningHead」という名前の要素変数に設定します。<sectionTitlesMultiline>要素は、それが発生したドキュメントから削除されます。

そして、要素変数は content：プロパティで使用できます。

```
  @page landscape:left {

    @top-left {
      content: element(runningHead start);
      border-bottom: 0.5pt solid black;
      margin-top: 0.25in;
      margin-bottom: 0.33in;
      vertical-align: bottom;
      text-align: left;
      font-size: 10pt;
    }

    ...
  }
```

ここで、content：プロパティは element（）関数を使用して「runningHead」要素変数の値を取得し、それを左上のページ領域のコンテンツとして使用します。

position：running（）を使用すると、適用する要素が消費されることに注意してください。これは、たとえば、柱（ランニングヘッダ）にタイトル要素またはタイトル要素コンテナを単純に反映できないことを意味します。メインフローと柱（ランニングヘッダ）のタイトルには、別個のソース要素が必要です。

たとえば、Municode HTML 前処理変換には、<sectionTitleMultiline>要素を生成する次のルールがあります。

```
  <xsl:template mode="frills-detailed" match="xhtml:header">
    <xsl:variable name="sectLevel" as="xs:integer"
      select="count(ancestor-or-self::xhtml:section)"
    />
    <sectionTitlesMultiline class="sect-{$sectLevel}">
      <!-- Process ancestors from highest to lowest -->
```

```
      <xsl:apply-templates select="(ancestor::xhtml:section)"
        mode="frills-get-multi-line-entries"
      />
    </sectionTitlesMultiline>
  </xsl:template>
```

作成されたソースは次のとおりです。

```
  <section id="x88B6B95C248C" data-type="section">
    <header>
      <h1>Sec. 1.1</h1>
      <p data-type="subtitle">Incorporation.</p>
    </header>
    ...
  </section>
```

そして変換結果は次のとおりです。

```
  <section id="x88B6B95C248C" data-type="section" class="">
    <header>
      <sectionNumberVerso>§ 1.1</sectionNumberVerso>
      <sectionNumberRecto>§ 1.1</sectionNumberRecto>
      <sectionTitlesMultiline class="sect-4">
        <headerLine class="charter">
          <h1 class="">Charter</h1>
        </headerLine>
        <headerLine class="chapter">
          <h1 class=" has-subtitle">Chapter 1</h1>
          <p data-type="subtitle" class="">General Provisions</p>
        </headerLine>
        <headerLine class="section">
          <h1 class=" has-subtitle">Sec. 1.1</h1>
          <p data-type="subtitle" class="">Incorporation.</p>
        </headerLine>
      </sectionTitlesMultiline>
      <h1 class=" has-subtitle">Sec. 1.1</h1>
      <p data-type="subtitle" class="">Incorporation.</p>
    </header>
```

ほとんどではありませんが、多くの出版物は文字列やページ端のコンテンツではなく要素を必要とします。そのためには、このタイプの追加マークアップを生成して提供する必要があります。

子孫または後続の要素プロパティに依存するスタイル設定

CSS セレクタは、カレントノードまたは前に出現したノードのみを選択でき、子孫または後続の要素を参照できません。

Municode コンテンツには、段落のスタイルがその子孫またはコンテンツに一部依存する少なくとも 2 つのケースがあり、<p>要素で特定のクラス値を設定する必要があります。

```
<xsl:template match="xhtml:p">

  <!-- This will always reflect the original @class value, if any -->
  <xsl:variable name="class-tokens" as="xs:string*">
    <xsl:apply-templates mode="set-class-for-p" select="."/>
  </xsl:variable>

  <xsl:copy>
    <xsl:apply-templates select="@* except (@class)" mode="#current"/>
    <xsl:if test="exists($class-tokens)">
      <xsl:attribute name="class"
        select="string-join($class-tokens, ' ')"
      />
    </xsl:if>
    <xsl:apply-templates select="node()" mode="#current"/>
  </xsl:copy>

</xsl:template>

<xsl:template mode="set-class-for-p" as="xs:string+" priority="10"
  match="xhtml:p[.//xhtml:span[exists(@data-lf)]]">
  <xsl:sequence select="'lf'"/>
  <xsl:next-match/>
</xsl:template>
```

```
<xsl:template mode="set-class-for-p" as="xs:string+"
   match="xhtml:p[ends-with(normalize-space(.),
':')][not(ancestor::xhtml:header)]
   ">

   <xsl:sequence select="'keepwithnext'"/>
   <xsl:next-match/>
</xsl:template>
```

コンテンツの合成または並べ替え

これは、要求された体裁を達成するために、要求通りの適切な構造を単に構築することです。これは CSS の制限に対する回避策ではなく、単なる要件です。多くの出版物では、デジタル配信にも要件がある可能性があります。

CSS を使用して構築できない生成文字列

CSS の content：プロパティと：before および：after 擬似要素を組み合わせるとかなりのことができますが、CSS は文字列操作やより複雑な計算（たとえば、日付と時刻をフォーマット値に変換する）を行う機能をもっていません。したがって、純粋にスタイルだけの問題として生成されるテキストを含む属性、または要素を生成する必要がある場合があります。

CSS ページ組版とエリアツリー

Antenna House Formatter（AHF）は、構成されたページの XML 表現である「エリアツリー」を生成し出力できます。エリアツリーは、すべてのフォントの詳細、配置の詳細など、レンダリングされたページの生成に必要なすべての情報を取得します。AHF は入力としてエリアツリーを扱い、最終的な成果物、つまり PDF を生成できます。

エリアツリーを後処理するには、次の情報を含める必要があります。

- 各変更の開始と終了（Municode の用語では「take」）
- ページ番号を取得する必要のある ID を持つ、各要素の開始と終了（節、表、図）

- 前置文字列付きページ番号（prefolio）と後置文字列付きページ番号(postfolio)のテキストの区別を含み、ページの端の領域に表示されるページ番号。
- その他の重要な要素の境界または出現。例えば更新の説明、ページごとにカウントする必要があるもの（テーブル、画像など）。

AHF は、エリアツリーに任意の情報を差し入れる一般的な方法をもっていません。ただし、入力要素で見つかった@id 値は保持されます。

構造化フィールドである@id 値を使用して入力 HTML の要素を構築することにより、これを活用します。

```
<areaTreeMarker id="take:take-begin:job=S138-U02:d20p60"/>
```

それは、このエリアツリー要素になります。

```
<InlineArea id="take:take-begin:job=S138-U02:d20p60"
  font-size="0pt"
  width="0pt"
  height="0pt"
  baseline-after="0pt"
  ...
/>
```

InlineArea には幅と高さがゼロの有効サイズがあるため、InlineArea が発生するページのレンダリングには影響しないことに注意が必要です。

これらのエリアツリーマーカー要素は、HTML の前処理ステップで追加されます。

ページ番号については、ゼロ幅スペース文字（¥u200B）を使用して、ページ番号が発生する文脈でレンダリングされるページ番号の前置文字列付きページ番号（prefolio）、ページ番号（folio）、および後置文字列付きページ番号（postfolio）コンポーネントを分離します。

```
 @page :blank {
  size:     8.5in 11in;
  margin-left: 7.5pc;
  margin-right: 7.5pc;
  margin-bottom: 6pc;
  margin-top: 8pc;

  @bottom-center {
    content: string(prefolio, first) '¥200B' counter(page) '¥200B' string(postfolio, first);
```

```
    margin-top: 1pc;
    vertical-align: top;
    font-family: "New Century Schoolbook", serif, 'Arial Unicode';
    font-size: 10pt;
  }

  ...

}
```

ゼロ幅スペース文字は、このコンテンツの他の場所では使用されず、レンダリングされたページで見えることはありません。

エリアツリー内のページ番号の発生は、確実に見つけることができます。

```
  <!-- Gets all the text areas for the page's page number, including the prefolio and postfolio, if
       present.
  -->
  <xsl:function name="at:get-page-number-text-areas" as="element(at:TextArea)*">
    <xsl:param name="context" as="element()?"/><!-- Any element that is or is within a page viewport area -->

    <xsl:variable name="page" select="$context/ancestor-or-self::at:PageViewportArea"/>
    <xsl:variable name="margin-region" as="element(at:MarginRegionViewportArea)?"

    select="$page/at:PageReferenceArea/at:MarginRegionViewportArea[.//at:TextArea[@text = '&#x200b;']]"
    />
    <!-- At least in the legacy style there should be exactly one block with exactly one line with three or more text areas. -->
    <xsl:variable name="result" as="element(at:TextArea)*" select="$margin-region//at:TextArea"/>
    <xsl:sequence select="$result"/>
  </xsl:function>
```

もう一つの課題は、表示とは別に、各ページのページ番号の数値を記録することです。

XSL-FO では、ページのページ番号はページフォーマッティングオブジェクトのプロパティであり、AHF はエリアツリーにある序数のページ番号を記録します。

ただし CSS では、ページ番号は他のカウンターと区別できない単なるカウンターであるため、AHF はページ番号を記録しません。

ページ番号を正しく計算できるように、数値のページ番号が必要です。特定のページに表示ページ番号がない場合や、表示ページ番号が数字以外（ローマ字、アルファベットなど）である場合があります。したがって、ページごとに、数値の（序数）ページ番号、表示されるページ番号、および正式なページ番号形式（ローマ字、アラビア語など）を知る必要があります。特に、ローマ数字は、ローマ数字で使用されるアルファベットのページ番号と区別できないため、ページ番号自体からページ番号の形式を判別することはできません。

この情報を取得するために、コーナー領域を使用します。コーナー領域は、Municode ページレイアウトでは使用されません。

CSS には 4 つのコーナー領域があり、各ページの端には 3 つの端領域があります。

数値のページ番号とページ番号の形式は、次のようなコーナー領域で取得されます。

```
@page {
  size:     8.5in 11in;
  margin-left: 7.5pc;
  margin-right: 7.5pc;
  counter-reset: footnote;
  background-image: attr(background-graphic, url);
  background-repeat: no-repeat;
  background-size: contain;
  background-clip: border-box;
  background-position: 50% 50%;

  @bottom-left-corner {
    content: '^npm^' counter(page);
    visibility: hidden;
    font-size: 8pt;
    color: white;
    font-family: monospace, 'Arial Unicode';
  }

  @bottom-right-corner {
```

```
      content: '^pnf:1';
      visibility: hidden;
      font-size: 8pt;
      font-family: monospace, 'Arial Unicode';
    }

    ...

}
```

このページルールはすべてのページに適用されることに注意してください（ページ名修飾子はありません）。「hidden」の visibility 値は、テキストがエリアツリーにあるが、実際のページには表示されないことを意味します。

その結果、エリアツリー要素は次のようになります。

```
<MarginRegionViewportArea region-name="bottom-left-corner"
  visibility="hidden"
  ...>
  <MarginRegionReferenceArea ...>
    <BlockArea ...>
      <LineArea >
        <TextArea ...
          text="^npm^"
        />
        <TextArea
          text="1" ...
        />
      </LineArea>
    </BlockArea>
  </MarginRegionReferenceArea>
</MarginRegionViewportArea>
```

コーナー領域の書式設定により、それらは非表示になります（visibility="hidden"）が、後処理中に簡単に見つけることができます。同様の手法を使用して、ページ番号の形式を取得できます。

端領域は、デバッグやその他の目的にも使用されます。たとえば、Municode は、文書を草案で印刷する際に、ソースに関する詳細を端領域に配置します。

デバッグの補助として、CSS を端領域のページマスター名を反映するようにすばやく変更できま

す。

```
@page :left {
  size:     8.5in 11in;
  margin-left: 7.5pc;
  margin-right: 7.5pc;
  margin-bottom: 6pc;
  margin-top: 8pc;

  /*    Used for debugging page rule application. */
  @left-bottom {
    content: 'Page Rule: :left';
/*      content: none;*/
    font-size: 10pt;
    font-family: "Courier New", monospace, 'Arial Unicode';
    color: white;
    -ah-reference-orientation: 90;
    width: 4pc;
    height: 6in;
  }

  ...

}
```

各ページルールで color プロパティを「white」からたとえば「cyan」に変更することで、ページルール名がすべてのページに表示されデバッグに役立ちます。

したがって、入力 HTML（<areaTreeMarker>）に追加された要素、構造化されたコンテンツ（ページ番号を表示）、ページ端領域の使用と乱用、および AHF の要素 ID を自動取得する機能の組み合わせにより、エリアツリーの後処理をサポートするために必要な情報をエリアツリーに挿入することができます。

エリアツリーの変更

最終的な目標は、必要なページ（表紙、目次、更新の説明など）および新しく追加されたポイントペ

ージ番号により変更されたページ番号を反映した、更新パッケージを表すエリアツリーを作成することです。さらに、変更されたページのすべてのシーケンスには偶数ページが必要です。Munico de の編集規則では、変更セットは常に奇数（右側）ページで開始されるため、偶数（左側）ページで終了する必要があります。

Munico de は、変更されたページのセットを「takes」と呼ぶため、ソースで変更の境界をマークするために使用されるマーカーは「take markers」です。「take」という用語は、この節のコード例に反映されています。

エリアツリーの処理は、以下の段階を踏んで論理パイプラインとして実装されます。

1. ページ番号と形式を設定する
   序数のページ番号とページ番号形式を使用して、各ページビューポートエリア要素の属性を更新します。
   また、変更マーカーで指定された「ページ開始」値を適用します。（変更ページの自動ページ番号付けに関係なく、変更マーカーは変更セットの最初のページの実際のページ番号を指定できます。）変更マーカーは HTML 階層内のどこでも発生する可能性があり、新しいページシーケンスを採用するために簡単に使用できないため、初期のページ組版スタイルは新しいページシーケンスを作成しません。そのため、事後にエリアツリーのページ番号を更新する方が簡単です。
   このステージのこの出力は、ページ番号の詳細の信頼性が高く、後続の処理で簡単に取得できるエリアツリーです。
   デバッグのために、結果のエリアツリーを検査して、ページ番号が正しく設定されていることを確認することもできます。

2. ページ番号を更新する
   ページが出力結果ページ（必要とされるページまたは変更セットのページ）にあるかそうでないかをマークします。
   変更セットにあるページの場合、表示ページ番号を更新して必要なポイントページを反映します。
   変更セットに必要な、空白の偶数ページを生成します。そのページは偶数ページで終了せず、または次のページがまだ空白でない必要があります。

3. ページをフィルターする
   更新パッケージが要求された場合、更新パッケージに存在するとマークされていないすべてのページを除外します。

4. 絶対ページ番号を振り直す
   AHF は各ページの絶対ページ番号を取得し、それらが正しい番号である必要があるため、フィルタリング後の絶対ページ番号を反映するためにページの番号を振り直す必要がありま

す。

5. 最終更新処理
   最終的な表示ページ番号を反映するために、主にページ番号参照を更新して、最終的な表示ページ番号を反映します。また、後処理には必要だったが、非表示のコーナー領域など、顧客に提供される最終 PDF には不要または必要とされないものをエリアツリー内で除外します。PDF のユーザーは多くの場合、新しい条例を起草するために PDF からカットアンドペーストするため、非表示のコンテンツをまでコピーされてしまうことは良くありません。

6. ページ番号データベースを更新する
   クライアントに配信されるパッケージを意味する「最終」更新パッケージを作成する時、ページ番号データベースが更新され、出版物の新規および変更されたページの詳細が反映されます。

ページ番号と形式を設定する

このフェーズでは、ページごとに現在のページシーケンスの序数ページ番号と、その番号の表示形式（アラビア語、ローマ字など）を決定していきます。

AHF は CSS モデルでは利用できないため、この情報を取得しません。

XSL-FO では、ページ番号はページシーケンスのプロパティとして明示的に定義され、必要に応じて専用のページ番号参照フォーマッティングオブジェクトを通じて反映されます。

一方 CSS では、ページ番号は他のカウンターと同様に単なるカウンターであり、ページまたはページシーケンスの明示的なプロパティではありません。CSS は新しいページごとに自動的に増分される「page」という名前の組み込みカウンターを定義しますが、これは単に利便性のためで、ページ番号を反映するために「page」カウンターを使用する必要はありません。

所定のページでは、通常のコンテンツプロパティのページ番号カウンターへの参照によって、ページ独自のページ番号が反映されます。

```
@page portrait-first:right {

  counter-reset: footnote;
  counter-reset: page 1;

  @bottom-center {
    content: string(prefolio, first) '¥200B' counter(page) '¥200B' string(postfolio, first);
```

```
    margin-top: 1pc;
    vertical-align: top;
    font-family: "New Century Schoolbook", serif, 'Arial Unicode';
    font-size: 10pt;
  }
  ...
}
```

したがって、CSS プロセッサには所定のページの意図するページ番号を知るための信頼できる方法はありません。

AHF エリアツリーマークアップには、pageViewportArea 要素のページ番号の属性が含まれます。

```
<PageViewportArea
    ...
    abs-page-number="1"
    page-number="1"
    format="1"
    >
```

ページ番号属性には値がありますが、信頼性はありません。

CSS のこの制限を回避するため、CSS は序数のページ番号とフォーマット値を非表示テキストとしてコーナー領域に配置します。

```
@page {
  size:     8.5in 11in;
  margin-left: 7.5pc;
  margin-right: 7.5pc;
  counter-reset: footnote;
  background-image: attr(background-graphic, url);
  background-repeat: no-repeat;
  background-size: contain;
  background-clip: border-box;
  background-position: 50% 50%;

  @bottom-left-corner {
    content: '^npm^' counter(page);
    visibility: hidden;
```

```
      font-size: 8pt;
      color: white;
      font-family: monospace, 'Arial Unicode';
    }

    @bottom-right-corner {
      content: '^pnf:1';
      visibility: hidden;
      font-size: 8pt;
      font-family: monospace, 'Arial Unicode';
    }

  }
```

これにより、エリアツリー内で検索しやすいデータが得られ、個々からページ番号と形式を見つけて、pageViewportArea 要素に常に存在するかのように設定できます。

```
<MarginRegionViewportArea
    visibility="hidden"
    region-name="bottom-left-corner"
    ...
>
  <MarginRegionReferenceArea ...>
    <BlockArea ...>
      <LineArea ...>
        <TextArea
          text="^npm^"
          ...
        />
        <TextArea
          text="1"
          ...
        />
      </LineArea>
    </BlockArea>
  </MarginRegionReferenceArea>
</MarginRegionViewportArea>
```

pageViewportArea ページにページ番号とフォーマットを配置することは厳密には必要ではありませんが、後続処理がより簡単になり、特定のページのページ番号の詳細を繰り返し検索する必要がなくなります。

ページ番号を更新する

変更されたページの特定のシーケンスについて、シーケンス内の各ページのページ番号を更新して、変更開始処理命令で指定されている場合はシーケンスの初期ページ番号と、必要なポイントページ番号の両方を反映する必要があります。

ページ番号の更新には、3 つの主要な処理タスクが含まれます。

1. 変更ページセットの開始ページと終了ページを特定する。
2. 指定されている場合、開始ページ番号と変更セットに続くページのページ番号に基づいて新しいページ番号のシーケンスを作成する。（変更の後にページ番号がリセットされる新しいページシーケンスの開始が続く場合は、（例えば章の終わり）指定できない。）
3. 変更セットの各ページの表示ページ番号を更新する。これには、表示されるページ番号の幅の変更を反映するために、ページ上のページ番号の水平位置を調整することが含まれる。

変更ページセットを特定する

変更境界は、特定の構造を持つ ID を持つインラインオブジェクトによって、エリアツリーに反映されます。そのインラインオブジェクトは編集者によって挿入された変更マーキング処理命令（「takemarkers」）から HTML 前処理の一部として生成された areaTreeMarker 要素から作成されます。

作成されたソースは次のとおりです。

```
<?pdf take-begin job="S01" firstpage="15"?>
<section id="x88B6B95E7E6A" data-type="chapter" >
  ...
<?pdf take-end job="S01" firstpage-ref="15"?>
...
</section>
```

この結果、ページ組版処理において、この HTML 入力が行われます。

```
<areaTreeMarker id="take:take-begin:job=S01:firstpage=15:d16p512"/>
<section id="x88B6B95E7E6A" data-type="chapter" >
```

```
  ...
<areaTreeMarker id="take:take-end:job=S01:firstpage=15:d16p512"/>
...
</section>
```

そして、これらの要素がエリアツリーに反映されます。

```
<InlineArea
  id="take:take-begin:job=S01:firstpage=15:d16p512"
  width="0pt" height="0pt"
  ...
/>
...
<InlineArea
  id="take:take-end:job=S01:firstpage-ref=15:15:d16p512"
  width="0pt" height="0pt"
  ...
/>
```

これらのマーカー領域は、変更セット内にあるページを検索するユーティリティ関数によって使用されます。

```
<!-- Determine if the context element is the start of a take
       for the specified update ID. A given page can have zero or more take starts
       for a given update ID.
       @param context A PageViewPortArea element (or other element that
                      could be an ancestor of a take marker)
       @param jobID The update ID to check for
       @param doDebug Turns debugging on or off.
       @return true if a take begin marker is found for the specified update ID.
  -->
<xsl:function name="at:is-take-start" as="xs:boolean">
  <xsl:param name="context" as="element()"/>
  <xsl:param name="jobID" as="xs:string?"/>
  <xsl:param name="doDebug" as="xs:boolean"/>

  <xsl:variable name="page" as="element(at:PageViewportArea)"
       select="$context/ancestor-or-self::at:PageViewportArea"
  />
```

```
    <xsl:variable name="take-specifier" as="element()*"
      select="at:get-take-specifier($context, $jobID)"
    />
    <xsl:variable name="result" as="xs:boolean"
      select="exists($take-specifier)"
    />
    <xsl:sequence select="$result"/>
  </xsl:function>

  <!--
    Gets the element that specifies the start of a take for the
    the specified update.
    @context Element to look in for a take specifier
    @jobID The ID of the update to get the take specifier for
    @return The first element that starts a take for the specified update, if any.
    -->
  <xsl:function name="at:get-take-specifier" as="element()?">
    <xsl:param name="context" as="element()"/>
    <xsl:param name="jobID" as="xs:string?"/>

    <xsl:variable name="take-specifier" as="element()*"
      select="($context//*[starts-with(@id, 'take:take-begin:')]
                          [contains(@id, concat('job=', $jobID))])[1]"
    />

    <xsl:sequence select="$take-specifier"/>
  </xsl:function>
```

これらの関数により、変更されたページとされていないページを簡単に区別できます。

```
  <!-- Context is the area tree root element -->
  <xsl:template name="update-page-numbers">
    <xsl:param name="doDebug" as="xs:boolean" tunnel="yes" select="false()"/>

    <xsl:for-each-group
      select="at:PageViewportArea"
      group-starting-with="at:PageViewportArea[at:is-take-start(., $jobID)]">
```

```
        <xsl:choose>
          <xsl:when test="at:is-take-start(., $jobID)">
            <xsl:call-template name="update-page-numbers-for-take-group">
              <xsl:with-param name="doDebug" as="xs:boolean" select="$doDebug"/>
              <xsl:with-param name="pages" as="element(at:PageViewportArea)+"
                select="current-group()"
              />
            </xsl:call-template>
          </xsl:when>
          <xsl:otherwise>
            <!-- Must be before first take -->
            <xsl:apply-templates                                    select="current-group()"
mode="update-page-numbers">
              <xsl:with-param name="doDebug" as="xs:boolean" select="$doDebug"/>
            </xsl:apply-templates>
          <xsl:otherwise>
        </xsl:choose>
      </xsl:for-each-group>
    </xsl:template>
```

この時点で、変更されたページのセットの開始（the take start）はわかりますが、終了ページはわかりません。

変更されたページのセットは、常に偶数ページでなければなりません。

変更セットの明示的にマークされた最後のページが偶数である場合、それで完了です。

しかし、最後のページが偶数でない場合は、空白の偶数ページを変更シーケンスに追加する必要があります。

空白ページは、最後に変更されたページに続くページが偶数の空白のページ（たとえば、続くページを奇数ページにするために生成されたページ）である場合、エリアツリーから取得できます。それ以外の場合は空白のページを合成する必要があります。これは現在のページをコピーし、ページの絶対および序数ページ番号に 1 を追加することにより行われます。表示ページ番号を含むページ端領域や、保持する必要がある他の端領域コンポーネントなど、空白ページに必要な部分のみが保持されます。XSLT には、必要に応じてこれを調整する簡単な構成があります。これには、空白ページと非空白ページ、つまりページ本文の有効な文字列値が空白のみであるか、「このページは意図的に空白のままにしてある」と設定されたマーカーと一致するページを区別できる機能も必要です。[3]

もう一つの複雑な要因は、変更セットが最初に空白の奇数ページで終了する場合です。この場合、2 番目の空白の偶数ページを追加するのではなく、空白の奇数ページを単に省略します。3 つの空白ページで終わる変更セットを使用することは決してできません。編集者のエラーにより不要な改ページがされた場合、または他の要因の組み合わせによって 2 つの空白ページが生じ、2 つの空白ページで終わる可能性があります。

新しいページ番号シーケンスを構築する

変更セット内のページのセットが与えられると、セット内のページ数、開始ページ番号、および（もし存在すれば）変更セットに続くページのページ番号がわかります。

これにより、開始＋それに続くページと実際のページ数との差を計算する簡単なものになります。元のページの数よりも序数が大きいページは、ポイントページです。

次に、XSLT は新しいページ番号のシーケンスを作成し、それを使用して変更セット内のページを更新します。（簡潔にするために一部詳細を割愛しています。）

```
<xsl:template name="update-page-numbers-for-take-group">
  <xsl:param name="pages" as="element(at:PageViewportArea)+"/>

  <xsl:variable name="page-numbers" as="xs:string*">
    <xsl:variable name="first-page-number" as="xs:string?"
      select="at:get-folio($take-start)"
    />
    <!-- All pages should have page numbers of one form or another
         but it could happen that a page has no number -->
    <xsl:if test="exists($first-page-number)">
      <xsl:sequence select="at:calculate-page-numbers-for-take(
        $pages-in-take, $jobID)"
      />
    </xsl:if>
  </xsl:variable>

  <xsl:for-each select="$pages-in-take">
    <xsl:variable name="pos" as="xs:integer" select="position()"/>
    <xsl:variable name="new-page-folio" as="xs:string?"
      select="$page-numbers[$pos]"
    />
    <xsl:apply-templates select="." mode="update-page-numbers">
```

```
        <xsl:with-param name="new-page-folio" as="xs:string?"
          select="$new-page-folio"
        />
        <xsl:with-param name="in-job" as="xs:boolean" select="true()"/>
      </xsl:apply-templates>
    </xsl:for-each>
</xs:template>

...
<xsl:function name="at:calculate-page-numbers-for-take" as="xs:string+">
    <xsl:param name="pages" as="element()+"/>
    <xsl:param name="jobID" as="xs:string"/>

    <!-- The number of pages whose page number does not need to change -->
    <xsl:variable name="ordinal-page-count" as="xs:integer"
      select="$next-page-num - $first-page-num"
    />
    <xsl:variable name="point-page-count" as="xs:integer"
      select="count($pages) - $ordinal-page-count"
    />
    <xsl:variable name="point-page-count" as="xs:integer"
      select="if ($point-page-count lt 0)
                then 0
                else $point-page-count"
    />
    <!-- The number of pages whose page number does not need to change -->
    <xsl:variable name="ordinal-page-count" as="xs:integer"
      select="$next-page-num - $first-page-num"
    />
    <xsl:variable name="point-page-count" as="xs:integer"
      select="count($pages) - $ordinal-page-count"
    />
    <xsl:variable name="point-page-count" as="xs:integer"
      select="if ($point-page-count lt 0)
                then 0
                else $point-page-count"
```

```
    />
    <xsl:variable name="before-points" as="element()+"
      select="$pages[position() le $ordinal-page-count]"
    />
    <xsl:variable name="point-pages" as="element()*"
      select="$pages[position() gt $ordinal-page-count]"
    />
    <!-- Generate display page numbers for each page before the point pages -->
    <xsl:for-each select="$before-points">
      <xsl:variable name="page-num-base" select="string($first-page-num + position()
- 1)" as="xs:string"/>
      <xsl:number value="$page-num-base" format="{$page-number-format}"/>
    </xsl:for-each>
    <!-- Generate display page numbers for point pages -->
    <xsl:for-each select="$point-pages">
      <xsl:variable name="point-number" as="xs:integer" select="position()"/>
      <xsl:sequence select="concat($point-page-base-formatted, '.',
$point-number)"/>
    </xsl:for-each>
  </xsl:function>
```

この時点で、各変更セットの各ページの表示ページ番号は、開始ページ番号とポイントページ番号の適用を反映するように更新されています。これらのページへの参照はまだ更新されていません。

また、このプロセスでは、すべてのページを「ジョブ内」または「ジョブ外」としてマークし、フィルタリングステップで使用されます。変更セットの一部であるページ、または常に含まれるページ（カバーページ、挿入指示など）はすべてジョブ内としてマークされ、他のすべてのページはジョブ外としてマークされます。

コードは、ページ番号の配置調整（中央揃え、左揃え、または右揃えか）の仕方を知る方法がないという単純な理由で、ページ上のポイントページ番号の水平位置を調整しません。少なくともMunicode スタイルでは、ポイントページを追加したり、数字を 1 桁から 2 桁に変更したりする視覚的な影響は最小限であり、通常は気に留められません。

ただし、いくつかの文脈のページへの参照は、調整する必要があります。たとえば目次では、数字が常に右揃えであるため、水平方向の配置の変更が顕著になるためです。これらの文書は通常、フローテキストの参照にページ番号を使用しないため、問題が簡単になります。

ページ番号が使用された場合、おそらくページ番号参照の数値部分の周りに余分なスペースを確保するか、現在の位置合わせと行端揃えを示すために使用するマーカー技術が必要になるでしょう。その情報があれば、ページ番号の参照の前後にテキストの水平位置を調整しても問題はないはずです。最悪の場合、コードで単語や文字の間隔調整を適用して、長い数字で行が終了して余白を通過させないようにする必要があります。（たとえば、境界線が重なってしまうテキストや、ページ上の同じ行にあるとタグ付けされていないテキスト）

エリアツリーには、個々のテキスト文字列レベルまでのすべての幾何学的情報が含まれているため、常にレイアウトを調整したり、重複を検出したりすることは可能ですが、通常それほどのレベルの精巧さは必要ありません。

法律上の出版物では単純に問題を回避するために通常、節、段落、図または表番号（および場合によってはタイトル）のみを参照します。通常ページ番号は、目次や索引などの生成されたナビゲーション構造に限定されます。

ページをフィルターする

フィルターされた更新パッケージが要求された場合、更新に含まれていないページはフィルターで除外されます。これは、ジョブ内にあるとマークされていない PageViewportArea、つまり前のフェーズのすべてのページに設定されている値を省略するということです。

```
 <xsl:template name="process-takes">
    <xsl:param name="doDebug" as="xs:boolean" tunnel="yes" select="false()"/>

    <xsl:for-each-group select="at:PageViewportArea" group-adjacent="@in-job eq
'true'">
      <xsl:choose>
        <xsl:when test="self::at:PageViewportArea[@in-job eq 'true']">
          <xsl:for-each-group select="current-group()"
group-starting-with="*[at:is-take-start(., $jobID)]">
            <xsl:sequence select="current-group()"/>
          </xsl:for-each-group>
          <!-- A set of take pages -->
        </xsl:when>
        <xsl:otherwise>
          <!-- Not take pages, ignore. -->
        </xsl:otherwise>
      </xsl:choose>
```

```
    </xsl:for-each-group>

  </xsl:template>
```

このコードは for-each-group を使用します。これは初期ロジックのリファクタリングを反映したもので、単純ではなく実際にはグループ化が必要でした。これは、単純な apply-templates と 1 組のテンプレート（@in-job eq 'true'に一致するものとしないもの）を使用して実行できます。

絶対ページ番号を振り直す

このフェーズに入力されるエリアツリーは、レンダリングされるページの最終セットを反映しています。AHF 要件を満たすために、各ページの@absolute-page-number 属性は、ドキュメント内のページの順序位置を正しく反映する必要があります。

```
  <xsl:template mode="renumber-abs-pages" match="@abs-page-number">

    <xsl:variable name="pageNumber" as="xs:integer"
      select="count(../preceding-sibling::*) + 1"
    />
    <xsl:attribute name="{name(.)}" select="$pageNumber"/>
    <xsl:attribute name="orig-abs-page-number" select="."/>
  </xsl:template>
```

このモードの残りは単に PageViewportArea 要素を処理する通常の ID 変換処理です。

最終更新処理

最後の処理ステップはページ番号参照を更新して、更新されたページ番号を反映し、エリアツリーがフィルターされている場合、変更されていないページへの参照については、ページ履歴データベースからページ番号を反映します。

これには、元のページ番号からの幅の変更を考慮するために、ページ番号参照が発生する行のテキストの水平位置を調整する必要があります。これには、元のページ番号と新しいページ番号の表示幅の違いを知る必要があります。

変更されていないページのページ番号は、ページ履歴データベースの要素 ID からページ番号へのマッピングで取得されます。

ページ履歴データベース

概念的に、ページ履歴データベースは、2 つの別個のデータセットで構成されます。

- 物理ページ（奇数ページと偶数ページ）の記録。絶対ページ番号、表紙と裏ページのページ番号（folio）の詳細、（デバッグに役立つ）奇数ページの最初の行で発生する更新ごとにページ番号を取得します。
- 各要素 ID 取得の記録、要素が発生する更新ごとの記録、要素が開始するページの絶対ページ番号とページ番号（folio）の詳細の記録、および記録が作成されたときのタイムスタンプの記録。要素のタイトルも取得されます（ID を持つ要素にはタイトルが必要です。そうしないと、このコンテンツのページ番号参照の対象になりません）。

物理ページの記録により、更新の説明と有効なページのリストを生成できます。要素履歴の記録により、変更されていないページ上の要素へのページ番号参照を生成できます。

履歴データベースは、特定の文書に対して長期間保持され、特定の更新がクライアントへの配信用に発行されるときに更新されます。

ページ履歴データベースは、さまざまな方法で実装できます。このプロジェクトの場合、初期実装では文書のソースを保持するバージョン管理システムで文書のソースとともに保持される XML ファイルを使用します。このファイルは、エリアツリーの後処理中に読み取られ、最終的な出版物が作成されると更新されます。

より堅牢な実装では、専用のデータベースアプリケーションと REST サービスを使用してデータベースを管理できますが、このプロジェクトでは適用範囲外であり、その基盤は利用できませんでした。

エリアツリーで要素ターゲットの詳細を取得する

CSS では、ページ番号参照は特定の要素を適用範囲とするカウンター参照です。

```
div.toc-entry > span.page > a.body:before,
div.minitoc.pg div.minitoc-entry > span.page > a:before
{
    content: target-counter(attr(href url), page);
}
```

そしてエリアツリーの結果は、さらに何かが行われない限り、ただ結果的に生じるテキストになります。

<a>要素は、AHF -ah-link：拡張プロパティを使用してナビゲート可能なハイパーリンクとしてもレンダリングされます。

```
a[href]
{
  -ah-link: attr(href url)
}
```

ページ番号への参照をマークするために、HTML 前処理は、テキストをページ番号参照としてマークする構造化 ID 値を持つ span を追加します。

```
<div class="toc-entry">
  <span class="title">
    <a href="#x88B6B95B1C04">
      <span class="h1">
        <span class="uppercase">Officials </span> of the <span class="uppercase">Town of
          Lovettsville, Virginia At the Time of This Codification </span></span></a></span>
  <span class="page" id="page:d121e33">
    <span class="page-number-marker">&#x200B;</span>
    <a id="pageref:d121e36" class="frontmatter" href="#x88B6B95B1C04"></a>
    <span class="page-number-marker">&#x200B;</span>
  </span>
</div>
```

<a>要素の@id である「pageref: {generated-id} "は、ページ番号の参照としてコンテンツをマークする（folio のページ番号部分のみ）のに対し、「page」クラスを持つ span は、「page: {generated-id}」の構造化された@id 値を使用してコンテンツをページ番号（full folio）としてマークします。

これにより、エリアツリーで確実に検索可能なマークアップが作成されます。

```
<InlineArea
  id="page:d121e49"
  ...>
  <InlineArea ...>
    <TextArea ...
      text="&#x200B;"
    />
  </InlineArea>
  <InlineArea
```

```
      internal-destination="x88B6B95B4E50"
      id="pageref:d121e52"
      ...>
      <InlineArea ...>
        <TextArea ...
          text-width="5.37pt"
          text="v"
        />
      </InlineArea>
    </InlineArea>
    <InlineArea ...>
      <TextArea ...
        text="&#x200B;"
      />
    </InlineArea>
  </InlineArea>
```

外側の InlineArea はページ（full folio）として識別されます。&#x200B（ゼロ幅スペース）のコンテンツを持つ InlineArea 要素は、前置文字列付きページ番号（prefolio）、ページ番号、後置文字列付きページ番号（postfolio）の境界をマークします。この例には、prefolio も postfolio もありません。

この例では、ページ番号は@id "pageref：d12e52"で識別されます。TextArea は、テキスト（"v"）とレンダリングされた幅（"5.37pt"）の両方を指定することに注意してください。

さらにターゲット ID は、-ah-link：拡張プロパティから生じる InlineArea の@internal-destination 属性で AHF によって自動的に取得されます。

最後の課題は、そのページの新しいページ番号を取得するために参照される物理ページを見つけるために、エリアツリー内でターゲット要素の開始位置を見つけることです。AHF はエリアツリーの要素の ID を取得しますが、必ずしも正しい位置で取得するわけではありません。

したがって、HTML 前処理は、ページ番号の参照が行われる可能性のある要素の開始と終了を示すマーカーを生成します。（現在のコンテンツセットの節、図、表）

```
<section id="x88B6B95B4E50" ...>
  <header>
    ...
    <h1 style="line-height:2em;">
      <span class="uppercase">Boards and Commissions</span>
```

```
        <br /> of the <br />
        <span class="uppercase">Town of <br /> Lovettsville, Virginia</span></h1>
      <p data-type="startpage">7</p>
    </header>
    <areaTreeMarker

id="marker:section:officialsoriginal:start:narrow:startpage=7:id=x88B6B95B4E50"
    />
    ...
    <areaTreeMarker
      id="marker:section:officialsoriginal:end:narrow:id=x88B6B95B4E50"
    />
</section>
```

マーカーはヘッダの後に出力され、最終結果で節見出しが生成されることに注意してください。これにより、たとえばヘッダの CSS が改ページを作成したり、ルールの保持により見出しの前に強制的に改行が発生した場合など、エリアツリー内のマーカーが見出しと同じページ上に表示され、前のページには表示されません。

これにより、信頼できる節の開始と終了のマーカーとして機能するインラインエリアがエリアツリーに作成されます。ページ番号の更新には開始マーカーのみが必要ですが、目次、索引や、ページ数が必要な場所などある節が占めるページをカウントするには終了マーカーが必要です。たとえば、請求目的で編集されたページと生成されたページの数を区別するためなどです。

特定の対象となる要素の実際のページ番号を取得するために、コードは最初のページ更新処理で使用されたのと同じロジックを使用して、ページにあるレンダリングされたページ番号を見つけます。つまりエリアツリーにそのページがあり、そのページの表示ページ番号を取得するのに、適切なユーティリティ関数への単一の関数呼び出しを行います。

```
<xsl:variable name="target-page" as="element()?"
  select="at:get-referenced-page($context)"
/>
<xsl:choose>
  <xsl:when test="at:is-within-take($target-page)">
    <xsl:sequence select="at:get-folio($target-page)"/>
  </xsl:when>
  <xsl:otherwise>
    <xsl:sequence
```

```
select="at:get-stored-page-number-for-pageref($pageListingDoc,      $context,
false())"/>
  </xsl:otherwise>
</xsl:choose>
```

ページ番号参照を更新する

表示ページ番号は、それらを確実に見つけやすくする形式で構成されています。

```
<LineArea text-altitude="8.87939pt" ...>
  <TextArea text="&#x200B;" .../>
  <TextArea text="2" .../>
  <TextArea text="&#x200B;" .../>
</LineArea>
```

マーカーは、テキストにゼロ幅スペース（¥u200B）が含まれる TextArea 要素を含む LineArea です。ゼロ幅スペースは、このコンテンツのいずれにも使用されないことと、視覚効果がないため、理想的なマーカー文字を作成します。

XSLT は表示ページ番号を操作（検索、ページ番号(folio)部分の取得など）するヘルパー関数を提供します。

ページ番号を更新する際の実際の課題は、大抵の場合新しいページ番号の幅が元の幅と異なることです。つまり、幅の違いを考慮したテキストエリアの水平位置に調整する必要があります。

ただし後処理には、新しいページ番号のレンダリング幅を知るための直接的な方法がありません。

1 つの解決策は拡張機能と Java2D ライブラリなどを使用して、エリアツリーで使用可能なフォントの詳細を使用して文字をレンダリングすることです。

このプロジェクトでは、より強引で信頼性の高いアプローチを採用しました。すなわち、文書で使用されるすべてのフォントとサイズで、ページ番号に出現する可能性のあるすべての文字（0-9、a-z、A-Z、"."、"-"、”:”など）のインスタンスをエリアツリーに含めるというアプローチで、レンダリングされた文字の検索を可能にする形式で行いました。この「文字サンプル」のセットは、HTML 前処理一部として生成されます。

HTML は次のとおりです。

```
<char-samples id="util:char-samples">
  <char-sample class="body">
    <char-set class="number-set sz83" id="util:char-set:body-sz83">
```

```
<decimal>.</decimal><char>0</char><char>1</char><char>2</char><char>3</char
><char>4</char><char>5</char><char>6</char>
<char>7</char><char>8</char><char>9</char><char>a</char><char>b</char><ch
ar>c</char><char>d</char><char>e</char><char>f</char>
<char>g</char><char>h</char><char>i</char><char>j</char><char>k</char><char
>l</char><char>m</char><char>n</char><char>o</char>
<char>p</char><char>q</char><char>r</char><char>s</char><char>t</char><char
>u</char><char>v</char><char>w</char><char>x</char>
<char>y</char><char>z</char><char>A</char><char>B</char><char>C</char><ch
ar>D</char><char>E</char><char>F</char><char>G</char>
<char>H</char><char>I</char><char>J</char><char>K</char><char>L</char><cha
r>M</char><char>N</char><char>O</char><char>P</char>
<char>Q</char><char>R</char><char>S</char><char>T</char><char>U</char><c
har>V</char><char>W</char><char>X</char><char>Y</char>
<char>Z</char>
</char-set>
  ...
  </char-sample>
  ...
</char-samples>
```

各<char-set>は CSS のフォント設定クラスの 1 つを指定するため、フォントとサイズはエリアツリーに表示されます。これらの値は文字自体とともに、任意の文字の検索キーとして機能します。唯一のメンテナンスタスクは、生成される XSLT を CSS スタイルと同期させて、関連するすべてのフォントとサイズの違いが表示されるようにすることです。

エリアツリーマークアップは次のとおりです。

```
<BlockViewportArea id="util:char-samples" ...>
  <BlockArea
    id="util:char-set:body-sz83" ...>
    <LineArea ...>
      <InlineArea ...>
        <TextArea ...
          width="2.224pt"
          font-family="NewCenturySchlbk LT Std"
          font-size="8pt"
          text-width="2.224pt"
```

```
            text="."
          />
        </InlineArea>
        ...
      <LineArea>
    </BlockArea>
  </BlockViewportArea>
```

各ソース文字は個別の要素にあるため、エリアツリー内の各文字に対して 1 つのテキストエリアが存在することが保証されます。テキストエリアは、文字をポイント単位で正確な幅を指定します。

次にこの XSLT コードは、フォントの詳細を与えている文字の詳細を検索します。

```
    <xsl:key name="char-samples"
      match="at:BlockArea[starts-with(@id, 'util:char-set:')]//at:TextArea"
      use="at:make-char-sample-key(@font-family, @font-size, @text)"
    />

    <!-- Try to find a character sample for the specified font family, size, and text value.
  -->
    <xsl:function name="at:get-char-sample" as="element(at:TextArea)?">
      <xsl:param name="context" as="element()"/><!-- Area tree element -->
      <xsl:param name="font-family" as="xs:string"/>
      <xsl:param name="font-size" as="xs:string"/>
      <xsl:param name="text" as="xs:string"/>
      <xsl:variable name="key" as="xs:string"
        select="at:make-char-sample-key($font-family, $font-size, $text)"
      />
      <xsl:variable name="result" as="element()?"
        select="key('char-samples', $key, root($context))[1]"
      />
      <xsl:sequence select="$result"/>
    </xsl:function>

    <xsl:function name="at:make-char-sample-key" as="xs:string">
      <xsl:param name="font-family" as="xs:string"/>
      <xsl:param name="font-size" as="xs:string"/>
      <xsl:param name="text" as="xs:string"/>
```

```
    <xsl:variable name="key" select="string-join(($font-family, $font-size, $text), ':')"/>
    <xsl:sequence select="$key"/>
  </xsl:function>
```

次に、影響を受けるラインエリアのテキストエリアに適用する調整量を決定するための計算を行い、更新するだけです。以下はリーダーを使用する ToC エントリ内のページ参照の場合です。（最も一般的なケース）

```
<!-- Page references in lines with leaders (e.g., ToC pages). This will be the most common case.

     There should only be one page reference, which means we can simply calculate the adjustment to
     the leader and all inline and text nodes following the leader should be good to go.

     The prefolio and postfolio should always be the same.
-->
<xsl:template match="at:LineArea[at:is-within-page-reference-line(.)]">

  <xsl:variable name="pagerefs" as="element(at:InlineArea)+"
    select=".//at:InlineArea[starts-with(@id, 'pageref:')]"
  />
  <!-- There should be exactly one TextArea in the pageref InlineArea -->
  <xsl:variable name="pageref" as="element(at:InlineArea)"
    select="$pagerefs[1]"
  />
  <xsl:variable name="target-page" as="element()?"
    select="at:get-referenced-page($pageref)"
  />

  <xsl:variable name="page-ref-text-area" as="element(at:TextArea)?"
    select="($pageref//at:TextArea)[1]"
  />
  <xsl:variable name="orig-page-number" as="xs:string?"
    select="$page-ref-text-area/@text"
```

```
/>

<!-- NOTE: This is the formatted page number, e.g., 'xix', not 19 -->
<xsl:variable name="page-number" as="xs:string?"
    select="at:get-referenced-page-number($pageref)"
/>

<xsl:choose>
  <xsl:when test="empty($page-number)">
    <!-- Nothing to do, just use what we have -->
    <xsl:next-match/>
  </xsl:when>
  <xsl:otherwise>
    <xsl:variable name="current-width" as="xs:double"
      select="at:get-length-value($page-ref-text-area/@text-width)"
    />
    <xsl:variable name="new-number-width" as="xs:double"
      select="at:get-string-width($page-ref-text-area, $page-number)"
    />

    <xsl:variable name="width-difference" as="xs:double"
        select="$new-number-width - $current-width"
    />
    <xsl:choose>
      <xsl:when test="$width-difference ne 0.0">
        <!-- Now figure out how many dots to remove from the
             leader to account for the added space: -->
        <xsl:variable name="leader-area" as="element(at:LeaderArea)"
          select="$pageref/preceding::at:LeaderArea[1]"
        />

        <xsl:variable name="leader-string" as="xs:string"
          select="$leader-area/at:TextArea/@text"
        />
        <xsl:variable name="leader-width" as="xs:double"
          select="at:get-length-value($leader-area/at:TextArea/@text-width)"
```

```
/>
<xsl:variable name="dots" as="xs:string*"
   select="tokenize($leader-string, ' ')"
/>
<xsl:variable name="dot-width" as="xs:double"
   select="$leader-width div count($dots)"
/>
<xsl:variable name="sign" as="xs:integer"
   select="if ($width-difference lt 0) then 1 else -1"
/>
<xsl:variable name="dot-count-diff" as="xs:integer"
   select="if (number($dot-width))
              then (($width-difference idiv $dot-width) + 1) * $sign
              else 0
              "
/>
<xsl:variable name="dot-count" as="xs:integer"
   select="$dot-count-diff + count($dots)"
/>
<xsl:variable name="new-leader-string" as="xs:string?"
   select="string-join(for $n in 1 to $dot-count return $dots[1], ' ')"
/>
<xsl:variable name="new-leader-width" as="xs:double"
   select="$dot-width * $dot-count "
/>
<xsl:copy>
   <xsl:apply-templates select="@*"/>
   <xsl:apply-templates>
      <xsl:with-param name="new-leader-string" as="xs:string?"
         tunnel="yes" select="$new-leader-string"
      />
      <xsl:with-param name="new-leader-width" as="xs:double?"
         tunnel="yes" select="$new-leader-width"
      />
      <xsl:with-param name="left-adjust" as="xs:double"
         tunnel="yes" select="$width-difference"
```

```
                    />
                    <xsl:with-param name="page-ref" as="element(at:InlineArea)?"
                      tunnel="yes" select="$pageref"
                    />
                  </xsl:apply-templates>
                </xsl:copy>
              </xsl:when>
              <xsl:otherwise>
                <xsl:copy>
                  <xsl:apply-templates select="@*"/>
                  <xsl:apply-templates>
                    <xsl:with-param name="doDebug" as="xs:boolean"
                      tunnel="yes" select="$doDebug"
                    />
                    <xsl:with-param name="new-leader-string" as="xs:string?"
                      tunnel="yes" select="()"
                    />
                    <xsl:with-param name="new-leader-width" as="xs:double?"
                      tunnel="yes" select="()"
                    />
                    <xsl:with-param name="left-adjust" as="xs:double"
                      tunnel="yes" select="0.0"
                    />
                    <xsl:with-param name="page-ref" as="element(at:InlineArea)?"
                      tunnel="yes" select="$pageref"
                    />
                  </xsl:apply-templates>
                </xsl:copy>
              </xsl:otherwise>
            </xsl:choose>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:template>
```

ページ番号データベースを更新する

最後のタスクは、作成中のページが更新パッケージの最終配信バージョンである場合に、ページ番号データベースを更新することです。（ただしデータベースは、更新が発行されるまで、特定の更新に対していつでも更新できます。更新が発行された時点でその更新のエントリは修正され、変更すべきではありません。）

この処理では、ソースバージョン管理の作業コピーにある場所からページ履歴データベースを読み取り、新しいファイルを一時的な場所に書き込みます。これにより、XSLT の同じファイルに対する読み取りと書き込みの制限が回避されます。処理実行の一部としてページ履歴データベースの更新が要求された場合、別の処理スクリプトが新しいバージョンのページ履歴データベースを作業コピーにコピーします。

ページリストを更新するロジックは、物理ページエントリを作成または更新するために各ページを処理し、その後、要素の element-to-page エントリを作成または更新するために各マーク付き ID を処理するだけです。

処理は既存のページデータベースから開始し、既に反映されているすべてのページを処理してから、データベースに反映されていないページを処理します。既存のデータベースがない場合は、通常の処理を適用する前に新しいデータベースが合成されます。

デバッグのために、プロジェクトにはページ履歴データベースの HTML ビューを生成するシンプルなスタイルシートが含まれています。

ページ履歴データベースに関する最後の課題は、古い XPP システムから新しいシステムに移行する出版物用に、最初にデータベースを作成することです。

XPP データから物理ページの履歴を生成することは可能ですが、XPP バージョンの節、図、表を新しい XML バージョンが対応する要素に関連付けたり、XPP バージョンでこれらの要素が落ちたページ番号を取得する簡単な方法はありません。これは最初の要素履歴データベースに手動で入力する必要があることを意味し、ムダな手動プロセスです。それを自動化する方法があるかもしれませんが、プロジェクトの範囲ではそれらを探求することができませんでした。たとえば、最後に発行されたPDFを調べて、見出しやその他の位置的な手がかりで要素を関連付けることができます。また PDF からエリアツリーを生成し、それらのエリアツリーに必要な開始マーカーと終了マーカーを入力する可能性も検討しました。しかし、そのレベルの自動化を考えると、関連付けが正しいことを検証するために必要な品質保証ステップがまだあります。幸いなことに、これは移行される文書ごとに 1 回限りのタスクです。

結論と今後の作業

変更されたコンテンツの開始と終了を識別するパブリケーションソース内のマークアップが与えられると、ページ組版に CSS を使用して、自動生成されたページ番号でルーズリーフ変更パッケージを生成することと、ページ組版エンジンによって生成された初期エリアツリーのポスト処理を実装することが可能です。

実装には、後処理に必要な情報をエリアツリーに取り込むためのいくつかのトリックを開発する必要がありましたが、結果のデータ処理はそれほど難しくありませんでした。通例の見切り発車で最終処理パイプラインを計算するのに、数回の繰り返し作業（イテレーション）が必要でしたが、最終ソリューションには問題自体の複雑さに見合ったレベルの複雑さがあるようです。

プロジェクトの計画には、変更の識別の自動化は含まれませんでした。DeltaXML などの XML 差分ツールを使用すると、2 つのエリアツリーを比較し、ページのシーケンスが変更されたことを判断し、必要に応じてレビューと調整のために適切なマーカーを文書ソースに戻すことができます。

もう一つ作業計画に含められる可能性があった事は、文書の完全な更新履歴を反映するエリアツリーへ、変更されたページを自動的に挿入することです。Municode は現在 PDF でこれを手動で行っていますが、必要に応じて PDF を生成できるエリアツリーを使用してこれを自動化することができるはずです。

参考資料

[AHF] Antenna House Formatter, Antenna House, Inc, http://antennahouse.com

[CSS] W3C CSS Working Group Standards and Drafts, https://www.w3.org/TR/#tr_Cascading_Style_Sheets_(CSS)_Working_Group

[CCSPAGE] Kimber, W. Eliot, *CSS Pagination Book, https://drmacro.github.io/css-pagination-book/, 2019*

---

[1] CSS ページ組版の概要については CCSPAGE を参照してください。

[2] 著者は XSL-FO 仕様への貢献者であり、本格的な生産を行うための世界標準を用いる古くからのユーザーの 1 人でした。（最初は Nokia が発行する携帯電話のマニュアルを、すべての言語で作成するために使用しました。）私は、今日でも多くの XSL-FO 実装を手掛けています。

ただし選択肢があれば、CSS ページ組版をオプションとして選択します。

残念ながら執筆時点では、Antenna House Formatter のみが CSS のページ組版の実装を十分に完了しており、市町村条例の要件など、私が通常扱うドキュメントの種類のレイアウトと印刷の要件を満たしています。これは、一部には XSL-FO の Apache FOP に匹敵する、CSS ページ組版用のオープンソースソリューションがないことを意味します。したがって、CSS ページ組版は、ページ文書の制作にとって魅力的なソリューションですが、商用ソフトウェアへの投資が必要です。商用ソフトウェアは非常に大きな価値を提供しますが、オープンソースソリューションがないため、CSS ページ組版を潜在的なソリューションとして提案することが難しいです。

[3] これを執筆している間に、「:blank」ページルールを使用して、ページを空白として明示的にマークすることも可能になると思います。