修 士 論 文

非XMLデータに対するXPath検索のための ラッパーのインターフェイスの設計

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

渡谷 賢治

2005年3月

修 士 論 文

非XMLデータに対するXPath検索のための ラッパーのインターフェイスの設計

指導教官　田島敬史 助教授

審査委員主査　田島敬史 助教授
審査委員　大堀淳 教授
審査委員　片山卓也 教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

310124 渡谷 賢治

提出年月: 2005 年 2 月

概 要

本論文では各種の非 XML データに対して XPath による検索を行うシステムを提案する．このシステムは，ある共通インターフェイスとなる操作群を各データ形式上で実現するラッパーと，ラッパーが提供する操作群を使って XPath を評価する共通モジュールから構成される．ラッパーが提供する操作群が低レベル過ぎると実行効率が悪くなり，逆に高レベル過ぎると各データ形式毎にラッパーを用意するコストが高くなる．本論文では，どのような操作群が XPath の効率的評価に必要か考察し，また，低レベルなインターフェイスの例である SAX インターフェイスを用いる場合や，より高レベルなインターフェイスを用いる場合との性能比較を行う．

目 次

第1章　はじめに

1.1　研究の背景と目的

近年，二次記憶の低価格化により，個人の計算機上やインターネット上の計算機上に様々なデータ形式の膨大な量のデータが保存されるようになっている．そのため，これらの雑多なデータ形式のデータ群から一元的に必要なデータを検索する機能に対する需要が高まっている．現在は，HTML，PDF，Microsoft PowerPoint などのデータ形式に対して，一元的にキーワードによる検索を行うシステムが実現されているが，データが膨大になり興味のあるデータの絞り込みが困難になるにつれて，単なるキーワードによる検索だけでなく，各データ中の構造を利用した「構造検索」が行えることが重要になっている．

この問題に対するアプローチとして，現在，汎用の標準データ形式として XML を用いることが提案されている．XML 形式を採用する利点としては，様々なデータに対して XML 用のツールを用いて統一的な処理を行えるという点がある．例えば，自分のアドレス帳，スケジュール表，メイルボックス等を XML 形式で保存しておけば，これらのデータに対して XML 用の検索言語である XPath[3] で一括して検索を行うなどの処理が可能となる．しかし，各アプリケーション専用の様々なデータ形式や画像等の様々なデータ種別毎の標準データ形式も依然として多く使用されており，データ量や性能上の問題から，今後も全てのデータが XML 形式になるとは考えにくい．そこで，本研究では，XML データと非 XML データの両方を統一的に処理するためのシステムの開発を行う．

非 XML データに対しての問い合わせには，一般に XML データへの問い合わせとして良く使用される XPath を用いる．XPath を用いた問い合わせを実現する手法としては，以下のような手法が考えられる．

1. データを XML に変換してファイルに保存し，検索時にそのファイルを用いる
2. データを XML 形式に変換し，それを RDB に格納し，この RDB に対して問い合わせを発行する
3. データを検索時にメモリ上に XML に変換して展開し，そのデータに対して検索処理を行う
4. 与えられた検索式の評価に必要な操作を，対象となるデータ形式上の操作に変換し，元データ上で検索処理を行う

上記 1 の手法の問題点は，XML に変換済みのデータと，XML に変換する前のデータの二つが存在することである．このため，データに変更があった場合には変換済みデータと元データとの間で整合性が取れていないという問題がある．また，二つのデータがシステム上に存在することによって，システムのデータ領域を余計に消費するという問題がある．

上記 2 の手法の問題点は上記 1 の手法と同様に，データの重複による，更新時の整合性の問題とデータ記憶領域の増加の問題があり，さらにデータの更新がある度に RDB に格納する為の処理時間のオーバーヘッドも問題となる．

上記 3 の手法の問題点は，メモリ上に XML 形式でデータを展開するとメモリ領域を大量に消費する問題がある．これは，XML の要素一つ一つを全てオブジェクトとしてメモリに展開する為に生じ，XML のフォーマットにもよるが，およそ 1MByte の XML データをメモリ上で展開すると 10MByte ほどメモリ領域を占有する．そのため，使用可能なメモリのサイズに比べて，扱う XML データのサイズが大きい場合には，この手法は適さない．

上記のことを踏まえ，本研究では上記 4 の方式を採用する．この方式では，データの重複は存在せず，またデータを XML としてメモリ上に全て展開しないため，メモリを大量に消費するという問題もない．

1.2 システムの構成

本研究で提案するシステムは，データ形式に依存しない処理を行う共通モジュールと，データ形式依存の処理を行うラッパーからなる．共通モジュールとラッパーが協調して動作する手順を以下に示す．

1. 共通モジュールは，ユーザから XPath 式を受取り，これを解析する．
2. 問い合わせの対象となる各データ毎に，そのデータ形式に応じたラッパーが起動される．
3. 共通モジュールは解析結果に基づいて，各ラッパーに対して，各データに対する操作命令を出す．
4. 各ラッパーは操作命令を各データ形式上の操作に変換し，各データのファイルにアクセスして必要な部分を読み出して共通モジュールに返すか，操作完了通知を返す．
5. 共通モジュールは読み出したデータの内容に基づいて，必要があればさらにラッパーに対する操作命令を出す．以下，最終結果が得られるまで，これを繰り返す．

また，以上の協調動作の様子を図示したものを図 1.1 に示す．

共通モジュールとラッパーは上記のような手順で協調動作を行うが，その役割の分担の仕方には，両者の間のインターフェイスとなる操作命令のレベルに応じて様々な形が考え

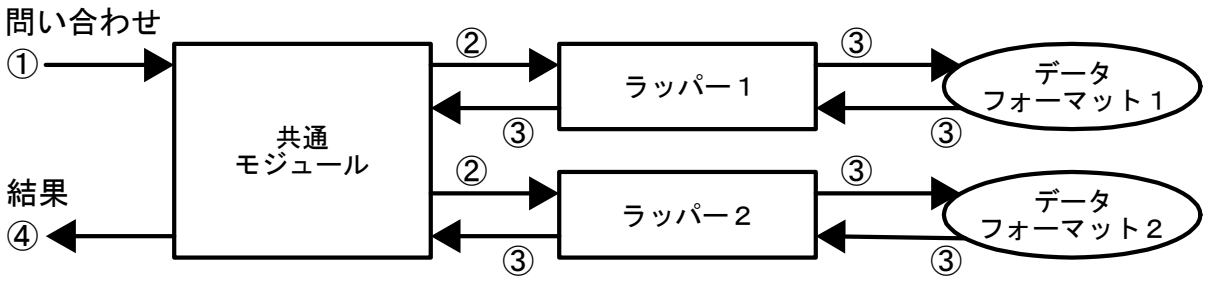

図 1.1: 共通モジュールとラッパーの構成

られる．ラッパーが共通モジュールに提供するインターフェイスの操作群が低レベル過ぎると，各ラッパーの作成は容易になるものの，与えられた XPath の実行の際に，各データ形式毎に依存した様々な最適化を行うことが困難になるため，実行効率が悪くなる可能性がある．一方，ラッパーが提供する操作群が逆に高レベル過ぎると，各ラッパー中に，データ形式に依存した最適化のロジックを組み込むことは容易になるものの，各データ形式毎にラッパーを作成するプログラミングのコストが高くなり，より多くのデータ形式をサポートするという上では障害となる．

ここでは，大きく分けて，以下の 4 方式を考えた．

方式 1: 共通モジュールは具体的な XPath の処理は行わなず，単に，対象データに応じたラッパーを起動して，ラッパーに XPath 式を渡す．ラッパーは XPath 式を処理してデータを操作し，その結果を共通モジュールに返す．すなわち，与えられた XPath 式に依存する処理は，全てラッパー側で行われ，共通モジュールは与えられた XPath 式に依存せずに常に同じ動作を行う．この方式ではラッパーに各データ形式固有の処理以外の処理 (XPath 式の評価) も含むため，ラッパーのコーディング量が増える．

方式 2: 共通モジュールは適切なラッパーを起動するとともに，XPath 式を解析し，単純な XPath 式の集合 (より具体的には，simple path の集合．simple path については詳しくは後述) に分割する．当初の XPath 式を分割して得られた各 XPath 式は，それぞれ，その操作を実現するためのファイルポインタの移動命令に変換されてラッパーに渡される．ラッパーは渡された命令に従いファイルポインタの移動 (あるいは移動先のデータの読み出し) を行い，操作が完了すると共通モジュールに通知し，次の命令を待つ．この場合，一つの操作命令が，XML 木上での単純なナビゲーションの１ステップを複数まとめたものに相当する．

方式 3: 共通モジュールは適切なラッパーを起動するとともに，XPath 式を解析し，その結果に基づいて，ラッパーにファイルポインタの移動命令を出す．一つの移動命令が XPath の一ステップ (XPath 中の「ステップ」については，詳しくは後述) に対応する．ラッパーは渡された命令に従いファイルポインタの移動 (あるいはさらに移動先のデータの読み出し) を行い，操作が完了すると共通モジュールに通知し，次

の命令を待つ．この場合，一つの操作命令が，XML 木上での単純なナビゲーションの一ステップに相当する．

方式 4: 共通モジュールは適切なラッパーを起動するとともに，XPath を評価する．ラッパーは，与えられた XPath 式依存の処理は一切行わず，常に同じ動作を行う．すなわち，常にデータの先頭から最後までの全体を処理して，データ全体の仮想的な XML ビューを共通モジュールに提供する．

方式 1 から 4 を比較すると，共通モジュールの作製は XPath 処理エンジンの実装を含む方式 4 は難しくなる．また，ラッパーの作成は，ラッパー毎に XPath の処理を行う部分のプログラミングが必要となるため方式 1 が難しくなる．すなわち，より一般的に言うと，ラッパーの提供するインターフェイスが低レベルであるほど，共通モジュールの作成は難しくなるが，ラッパーの作成は容易になり，逆にラッパーの提供するインターフェイスが高レベルであるほど，共通モジュールの作成は容易になるが，ラッパーの作成は難しくなる．共通モジュールは一つ作成すればよいのに対し，ラッパーは各データ形式毎に作成する必要があることを考えれば，ラッパーの作成コストが低い方が望ましいので，その意味では方式 4 の方が望ましい．

一方，検索効率は，方式 4 が最も悪く，方式 1 に近づくほど，効率化の余地がある．これは，前述のように，ラッパーが共通モジュールに提供するインターフェイスの操作群が低レベル過ぎると，与えられた XPath の実行の際に，各データ形式毎に依存した様々な最適化を行うことが困難になり，逆にラッパーが提供するインターフェイスのレベルが高レベルであると，ラッパー内に，各データ形式毎に依存したより広範な最適化を組み込むことが可能になるためである．

例えば，方式 4 において，ラッパーのインターフェイスとして，XML データへのアクセスの規格の一つである SAX インターフェイスを採用する場合を考える．SAX インターフェイスでは，与えられた XML データを先頭からスキャンし，データ中に出現するエレメントの開始タグや終了タグなどの出現を表すイベントの列をアプリケーションに出力する．よって，ラッパーのインターフェイスとして SAX インターフェイスを採用した場合，各ラッパーは，与えられた XPath 式に関係なく，そのデータ形式のデータ全体を XML に変換し，その XML 中の全ての要素を順に出力することになる．このようにした場合，巨大なデータから，その最初の方の一部のみを取り出すような検索において，共通モジュールによる XPath の処理において，データ中の不必要な部分の XML への変換を省略したくても，ラッパーは全てのデータを XML に変換してしまうことになり，XPath の処理効率は大幅に低下する．

そこで本研究では，中間位置である方式 2 と方式 3 を比較し，どちらがラッパーの作成コストが比較的低く，かつ十分な効率化が実現できるかを実験し，共通モジュールとラッパーの間のインターフェイスにどちらが適当かを判別する．

第2章　基礎的な事項と関連研究

2.1　XML

XML 形式のデータはノードにラベルのついた木構造で表現することができる．例として，図 2.1 の XML データを木構造にマッピングすると図 2.2 のような木構造になる．ここで，図 2.2 の楕円のノードは XML の要素ノードであり．矩形のノードはテキスト要素ノードである．テキストノード”#TEXT:”はテキスト値が空のテキストであり，”#TEXT:Alan”はテキスト値が”Alan”のテキストであることを示している．

2.2　XPath

XPath は XML データの木構造に対して問い合わせをする木パターン言語であり，先ほどの例である図 2.1 のような木構造を持つ XML データに対して，root の下の person の下に現れる name の値を XPath 式を用いて取得するには/root/person/name という XPath 式を使用する．

```
<root>
  <person>
    <name>Alan</name>
    <age>20</age>
  </person>
  <person>
    <name>Dennis</name>
  </person>
</root>
```

図 2.1: XML データの例

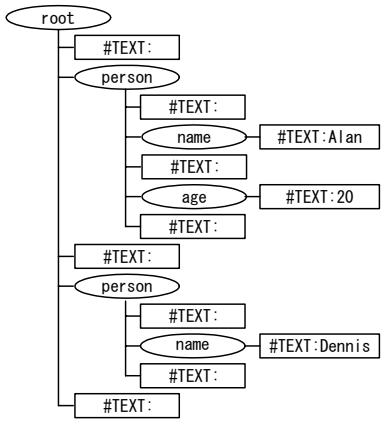

図 2.2: XML の木構造

表 2.1: ロケーションパスの構文

ロケーションパス	::=	相対ロケーションパス
		| 絶対ロケーションパス
絶対ロケーションパス	::=	”/” 相対ロケーションパス?
相対ロケーションパス	::=	ロケーションステップ
		| 相対ロケーションパス ”/” ロケーションステップ

2.2.1 ロケーションパス

XPath 式はロケーションパスという式で書かれ，絶対ロケーションパスと相対ロケーションパスで構成される．絶対及び相対ロケーションパスは”/”で区切られた 1 つ以上のロケーションステップ (後述) を並べたものである．ロケーションパスの構文を EBNF 式で書くと表 2.1 となる．この表に従うとロケーションパス/root/person[age]/name を以下の手順で分割することができ，最終的にロケーションステップとして root，person[age]，name が取得される．

1. /root/person[age]/name を ”/” と相対ロケーションパス root/person[age]/name に分割
2. root/person[age]/name を相対ロケーションパス root/person[age] と ”/” とロケーションステップ name に分割
3. root/person[age] を相対ロケーションパス root と ”/” とロケーションステップ person[age] に分割
4. root をロケーションステップ root とする

2.2.2 コンテキストノード

コンテキストノードとは処理対象のノードのノードである．図 2.1 で，XPath person/name/text() がある場合，この XPath を root 要素に対して処理を行う場合，root 要素をコンテキストノードと呼ぶ．

2.2.3 simple path

/root/person[name=”Alan”]/age を “[” と “]” で分割した部分式を simple path と呼び，上記の例では /root/person と name=”Alan” と /age に分割できる．simple path の例を表 2.2 に示す．

表 2.2: simple path の例

元の XPath 式	simple path 群
/root/person/name	/root/person/name
/root/person[name=”Alan”]/age	/root/person name=”Alan” /age
/root/person[name=”Alan”][age=”20”]/text()	/root/person name=”Alan” age=”20” text()

表 2.3: ロケーションステップの構文

ロケーションステップ　::=　axis ”::” ノードテスト predicate*

2.2.4　ロケーションステップ

節 2.2.1 で述べたように，XPath 式は”/”で区切られる文字列で分割することができる．これをロケーションステップと呼ぶ．ステップの構文を EBNF 式で書くと表 2.3 となる．この表で使われる axis，ノードテスト，predicate については後述する．

2.2.5　axis

axis はロケーションステップを使用して選択するノードとコンテキストノードとの間のツリー関係を指定する指示子である．/root/person[age]/name は省略された記述方式であり，省略せずに記述すると/child::root/child::person[age]/child::name となり，ロケーションステップに分割すると child::root，child::person[age]，child::name となり，それぞれのロケーションステップ中の child が axis である．

axis には child, descendant, parent, ancestor, attribute, namespace, preceding-sibling, following-sibling, self, ancestor-or-self, descendant-or-self, preceding, following の 13 種類がある．これらの中で良く使用されるものは child, parent, attribute, self, descendant-or-self の 5 種類である．この 5 種類には省略形が用意されており，axis の使用例とその説明及び省略形を表 2.4 に示す．

表 2.4: axis と命令との変換テーブル

XPath の axis の使用例	説明	省略形
chlid::x	あるコンテキストノードの子ノード x	x
parent::node()/child::x	あるコンテキストノードの親ノードの子ノード x	../x
attribute::x	コンテキストノードの属性ノード x	@x
self::node()/child::x	コンテキストノード自身の子ノード x	./x
descendant-or-self::node/child::x	コンテキストノード自身及びその子孫ノード x	//x

2.2.6 ノードテスト

節 2.2.5 で述べたように /root/person[age]/name を省略しないロケーションステップに分けると child::root，child::person[age]，child::name となる．ノードテストとはこれらの”::”の右側から”[”までの文字列である．この例では，それぞれ root，person，name がノードテストである．このように要素名を使用することもできるが，他に comment()，text()，processing-instruction()，node() という 4 種類を記述することができる．これらはノードの種類について規定しており，comment() はコメントノード，text() はテキストノード，processing-instruction() は XML 処理ノード，node() は全ノード種類を表す．よって，/root/person のテキストノードだけを取得するには/root/person/text() と書く．

2.2.7 predicate

predicate はロケーションステップ中に “[” と “]” で囲まれた部分を指し，child::person[age] の場合 age が predicate である．predicate は条件式であり，/root/person の子ノード name のテキストノードが Alan である場合の/root/person の子ノード age を取得する場合は，/root/person[name=”Alan”]/age と書く．

2.2.8 ステップ

本研究では，上記の用語以外にステップという単語を用いる．これは，ロケーションステップと predicate をまとめたもので，/root/person[age]/name の場合，/root，/person，[age]，/name がステップである．また，コンテキストノードを root 要素としたときの XPath 式 person を root 要素に対する相対ステップと呼ぶ．

2.3 DOMとSAX

この節ではXML データに対してアクセスするために一般に使用されるインターフェイスであるDOM とSAX について説明する．

2.3.1 DOM

DOM(Document Object Model) はW3C が提唱するプラットフォームや言語に依存しないインターフェイスであり，プログラムやスクリプトが文書の構造やスタイル，内容に動的にアクセスしたり更新することができる．現在DOM Level1, Level2, Level3 がW3C から提出されている．具体的には，XML 文書全体を木構造データとしてメモリ上に最初に展開し，その後ユーザは木構造上の現在位置ポインタをDOM が提供するインターフェイスによって移動させ，目的のノードへアクセスする．以下にDOM を用いて図2.2 の”Dennis”という文字列にアクセスする例を示す．

1. Document doc = XML 文書をオブジェクトDocument にマッピング;
   この処理はDOM に規定された処理ではなく，実装側で処理を行う
2. Node root = doc.getDocumentElement();
   root ノードを取得
3. NodeList root_children = root.getChildNodes();
   root ノードの子ノードを取得
4. Node person = root_children.item(3);
   root_children の4 個目のノード (person 要素ノード) を取得
5. NodeList person_children = root.getChildNodes();
   person ノードの子ノードを取得
6. Node name = person_children.item(1);
   person_children の2 個目のノード (name 要素ノード) を取得
7. Node text = name.getFirstChild();
   person_right の最初の子ノード (テキストノード) を取得
8. 文字列 string = text.getNodeValue();
   テキストノードの値 (文字列:Dennis) を取得

上の例のように，ラッパーによるデータのアクセスにDOM を用いる場合，ラッパーが最初 (1 行目) で非XML データを木構造データとしてメモリに展開する仕組みを作ることになる．このため，大容量データに対してDOM インターフェイスによるアクセスをすることが不可能である．

2.3.2 SAX

SAX は W3C が公式に提供するインターフェイスではなく，XML-DEV メーリングリストにより開発されたインターフェイスである．SAX 方式は，XML ファイルを先頭から読んでいき，開始タグ，終了タグ，文字列などが出現する度に，対応するイベントを発生し，そのイベントに関連づけられたハンドラを起動するというイベント駆動式のインターフェイスである．主に XML のノードを検索するために作られており，要素を追加したり更新するためには使用されないが，追加・更新をすることも可能である．また，XML のノードを検索するための検索指示子には主に XPath 式が用いられる．しかし，SAX はイベントを発生させるためだけのインターフェイスであり，XPath を用いた検索には別途 XPath 処理器が必要となる．SAX では，XML データ全体をメモリ上に展開するのではないため，大容量データに対して使用することが可能である．しかしファイルを先頭から一回走査する間に処理を行う方式であるため，ランダムアクセスには向かない．

2.4 関連研究

XML データではなく非 XML データを仮想的に XML として操作を行う枠組みに関する関連研究としては，既に BinX[1] や「バイナリデータ上の XML ビュー機構と XPath 処理の提案」[4] などがある．しかし，BinX は非 XML データとしてバイナリデータを対象としており，そのバイナリデータのデータがどのような値 (character，int，float 等) であるかや，バイトオーダ (リトルエンディアン，ビッグエンディアン) や，データの構造を規定した XML データを元にバイナリデータを仮想的に XML としてユーザに見せる．この際に使用される API は BinX 独自の物となる．

また,「バイナリデータ上の XML ビュー機構と XPath 処理の提案」は同じくバイナリデータに対し仮想的に XML として操作を行うが，これは XML 用のスキーマ言語である RELAX NG[5] を用い，汎用的なバイナリデータに対しその構造を記述し，DOM によるアクセスを行う研究である．

これらの研究は，データ形式の記述から自動的に，バイナリデータを仮想的に XML として扱う機構に関する研究であり，本研究の目的である，XPath による問い合わせのために非 XML データを仮想的に XML として扱うインターフェイスの設計とは，目的が異なる.「バイナリデータ上の XML ビュー機構と XPath 処理の提案」[4] では，XPath の効率的評価についても述べられているが，XML ビューのインターフェイスとしては上記のように，汎用性を重視して DOM を採用している．

第3章　共通モジュール

この章では各方式での共通モジュールの動作イメージについて説明する．

3.1　各方式の共通モジュール

この節では方式1〜4での共通モジュールについて説明する．

3.1.1　方式1の共通モジュール

方式1で用いる共通モジュールは単に起動するラッパーの種類を決め，そのラッパーを起動し，対象データとXPath式を渡す．共通モジュールからは対象データに対しXPath式を適用した結果が渡されるため，その結果をユーザに提供する．

3.1.2　方式2の共通モジュール

方式2で用いる共通モジュールは起動するラッパーの種類を決め，そのラッパーを起動し，対象データを渡した後XPath式をsimple pathに分解し，リストにする．その後リストに入っているsimple pathをある要素からほかの要素に移動するという移動命令や，ある要素を取得するという取得命令に変換しラッパーに命令をわたす．ラッパーからは命令の処理結果が返るのでその結果に基づき次の命令をラッパーに渡す．最終的にリストに入っている全ての命令がラッパー側に提示され，最後の命令がラッパーに受け入れられると，その結果をユーザに提示し，全ての命令が渡されずにファイルの最後まで行き着いたり，それ以降の命令が確実に失敗に終わると十分に判断される場合ユーザに失敗通知を提示する．

例として，ユーザからXPath式として/a/b/c[d/e]/fが提示されたとする．これを後述する，節4.1.2の表4.1に示すアルゴリズムで命令リストに変換すると，リストの要素は{”go /”, ”go a/b/c”, ”test d/e”, ”get f”}となる．この”go /” や ”go a/b/c”は移動命令で，”test d/e”はテスト命令と呼ばれ，”get f”は取得命令と呼ぶ．これをラッパーに先頭から順に提示し，最後の”get f”により/a/b/c[d/e]/fの要素を取得することができたら，その結果をユーザに示す．しかし，途中の ”test d/e”を評価している間にデータの最後が訪れたり，現在の位置以降にf要素が無いことがラッパー内で判断されたり，”go /”の結果

が偽となる場合はこの XPath 式が失敗したと判断され，ユーザに失敗したことが提示される．

3.1.3 方式 3 の共通モジュール

方式 3 で用いる共通モジュールは起動するラッパーの種類を決め，そのラッパーを起動し，対象データを渡した後 XPath 式をロケーションステップや predicate に分解し，リストにする．その後リストに入っているロケーションステップや predicate をある要素から他の要素に移動するという移動命令や，ある要素を取得するという取得命令に変換しラッパーに命令をわたす．ラッパーからは命令の処理結果が返るのでその結果に基づき次の命令をラッパーに渡す．最終的にリストに入っている全ての命令がラッパー側に提示され，最後の命令がラッパーに受け入れられると，その結果をユーザに提示し，全ての命令が渡されずにファイルの最後まで行き着いたり，それ以降の命令が確実に失敗に終わると十分に判断される場合ユーザに失敗通知を提示する．

例として，ユーザから XPath 式として/a/b/c[@d='e']/f が提示されたとする．これを表 4.2 に示すアルゴリズムで命令リストに変換すると，リストの要素は {"go a", "go b", "go c", "test @d", "get f"} となる．この"go a" や "go b"，"go c"は移動命令で，"test @d"はテスト命令と呼ばれ，"get f"は取得命令と呼ぶ．これをラッパーに先頭から順に提示し，最後の"get f"により/a/b/c[@d='e']/f の要素を取得することができたら，その結果をユーザに示す．しかし，途中の "test @d"を評価している間にデータの最後が訪れたり，現在の位置以降に f 要素が無いことがラッパー内で判断されたり，"go a"の結果が偽となる場合はこの XPath 式が失敗したと判断され，ユーザに失敗したことが提示される．

3.1.4 方式 4 の共通モジュール

方式 4 で用いる共通モジュールは起動するラッパーの種類を決め，そのラッパーを起動し，対象データを渡す．ラッパーからは SAX イベントが返るのでそのイベントに従い XPath 式を評価する．そして，ユーザにデータへの XPath 式の適用結果を提示する．

第4章　ラッパー

この章では各方式でのラッパーのインターフェイス及び動作イメージについて説明する．

4.1　各方式が提供するインターフェイス

この節では方式1〜4が提供するインターフェイスについて説明する．

4.1.1　方式1が提供するインターフェイス

方式1で用いるラッパーが提供するインターフェイスは簡潔であり，対象データと処理するXPath式を受け取り，共通モジュールにXPath式の処理結果を返すインターフェイスとなる．方式1内部でのデータアクセスにはDOM及びSAXを用いることができる．DOMを用いてXPathを処理するXPath処理機には Apache Xalan[2] や jaxen[6] 等があり，SAX を用いてXPathを処理するXPath処理機にはXSQ[7]等がある．

4.1.2　方式2が提供するインターフェイス

方式2で用いるラッパーが提供するインターフェイスは対象データを受け取り，その後simple pathを受け取り，その処理結果を返すインターフェイスとなる．内部でのデータアクセスには，効率性の面から，ランダムアクセスが求められる．

ラッパー内部での対象データへのアクセスにはメモリを大量に消費せず，かつ高速な方式を考案する．この方式ではSAXのように先頭からデータを見ていくが，現在いる要素からある関係にある要素に対応する位置まで移動してスキップしたり，戻ったりすることができる．これらの命令は，ラッパー内でsimple pathを処理した命令と，共通モジュールからの命令使用され，その処理結果を共通モジュール側に通知する．

使用することができる命令群について，以下に示す．

- **登録命令**
  - **regist x y fp**:simple path x と，そのコンテキストノード y と，そのファイルポインタ fp のセットをラッパーに登録する．登録されたらラッパーからクエリディスクリプタ qd が返される．

- **移動命令**
  - **go qd**:クエリディスクリプタ qd でラッパーに登録されたコンテキストノード y 以下において，qd でラッパーに登録された相対ステップ x の要素が存在するかどうかチェックする．この際，ファイルポインタはその要素が存在する場合，最初に現れるその要素の開始地点に移動する．
  - **go next qd**:クエリディスクリプタ qd でラッパーに登録されたコンテキストノード y 以下において，qd でラッパーに登録された相対ステップ x の要素が現在のノード X 以降に存在するかどうかチェックする．この際，ファイルポインタはその要素が存在する場合，最初に現れるその要素の開始地点に移動する．
- **テスト命令**
  - **text qd**:クエリディスクリプタ qd でラッパーに登録されたコンテキストノード y 以下において，qd でラッパーに登録された相対ステップ x の要素が存在するかどうかチェックする．この際，ファイルポインタはその要素が存在するかどうにかかわらず現在位置から移動しない．
  - **test next qd**:クエリディスクリプタ qd でラッパーに登録されたコンテキストノード y 以下において，qd でラッパーに登録された相対ステップ x の要素が現在のノード X 以降に存在するかどうかチェックする．この際，ファイルポインタはその要素が存在するかどうにかかわらず現在位置から移動しない．
- **取得命令**
  - **get qd**:クエリディスクリプタ qd でラッパーに登録されたコンテキストノード y 以下において，qd でラッパーに登録された相対ステップ x の要素が存在するかどうかチェックする．次に，その要素が存在する場合，その要素以下を取得する．
  - **get next qd**:クエリディスクリプタ qd でラッパーに登録されたコンテキストノード y 以下において，qd でラッパーに登録された相対ステップ x の要素が現在のノード X 以降に存在するかどうかチェックする．次に，その要素が存在する場合，その要素以下を取得する．

4.1.3 方式 3 が提供するインターフェイス

方式 3 で用いるラッパーが提供するインターフェイスは対象データを受け取り，その後命令群を受け，その処理結果を返すインターフェイスとなる．内部でのデータアクセスには，効率性の面から，ランダムアクセスが求められる．

ランダムアクセスをするためには，ラッパー内部にファイルポインタと呼ばれる数値を保持する．これは，記憶領域中で，次にデータを読み込む位置を示す数値であり，OS な

どに用意された API にこの数値を入れることで，次回データを読み込む位置を自由に変更できる仕組みである．

ラッパー内部での対象データへのアクセスにはメモリを大量に消費せず，かつ高速な方式を考案する．この方式では SAX のように先頭からデータを見ていくが，現在いる要素と指定した関係にある要素に対応する位置まで移動してスキップしたり，戻ったりすることができる．これらの命令は，XPath の一ステップに相当する物になっており，共通モジュールから命令が入力され，命令の処理結果を共通モジュール側に通知する．

使用することができる命令群について，以下に示す．

- **移動命令**
  - **go x**:現在のノード X の子ノード x が存在するかどうかチェックする．この際，ファイルポインタは子ノード x が存在する場合，最初に現れる子ノード x の開始地点に移動する．
  - **goAnyChild x**:現在のノード X の子孫ノード x が存在するかどうかチェックする．この際，ファイルポインタは子孫ノード x が存在する場合，最初に現れる子孫ノード x の開始地点に移動する．
  - **goNext x**:現在のノード以降の兄弟ノード x が存在するかどうかチェックする．この際，ファイルポインタは兄弟ノード x が存在する場合，最初に現れる兄弟ノード x の開始地点に移動する．
- **テスト命令**
  - **test x**:現在のノード X の子ノード x が存在するかどうかチェックする．この際，ファイルポインタは子ノード x が存在するかどうかにかかわらず現在位置から移動しない．
  - **testAnyChild x**:現在のノード X の子孫ノード x が存在するかどうかチェックする．この際，ファイルポインタは子孫ノード x が存在するかどうかにかかわらず現在位置から移動しない．
  - **testNext x**:現在のノード以降の兄弟ノード x が存在するかどうかチェックする．この際，ファイルポインタは兄弟ノード x が存在するかどうかにかかわらず現在位置から移動しない．
- **取得命令**
  - **get x**:現在のノード X の子ノード x が存在するかどうかチェックする．次に，子ノード x が存在する場合その子ノード x 以下を取得する．
  - **getAnyChild x**:現在のノード X の子孫ノード x が存在するかどうかチェックする．次に，子孫ノード x が存在する場合その子ノード x 以下を取得する．

– **getNext x**:現在のノード以降の兄弟ノード x が存在するかどうかチェックする．次に，兄弟ノード x が存在する場合その子ノード x 以下を取得する．

4.1.4 方式 4 が提供するインターフェイス

方式 4 で用いるラッパーが提供するインターフェイスは SAX のインターフェイスとなる．共通モジュール側からは対象データのみを受け取る．SAX イベントが発生したらそのイベントを共通モジュール側に通知する．

4.2 方式 2,3 の命令生成法

次に，方式 2,3 で使用される XPath 式を命令列に変換する法則について説明する．

4.2.1 方法 2 の命令生成法

方式 2 では共通モジュール側で XPath 式を simple path に分解し，その simple path を方式 2 のラッパーが提供するインターフェイスの命令に返還する必要があるが，その返還法を表 4.1 に疑似プログラムで示す．

表 4.1 の疑似言語を用いて /a//b[@c='d']/e を命令列に変換する例を以下に挙げる，

- リスト X は{ "/a//b","[@c='d']","/e" }となる
- 命令リストに"go /"を追加する
- x に"/a//b"が代入され，x は"/"から始まる文字列であり，x は X の最終要素では無いため，命令リストには "go a//b" が追加される
- 次のループでは x には"[@c='d']"が代入され，x は"["から始まる文字列のため Predicate 部であることがわかる．x は X の最終要素では無いため，"test @c='d'" を命令リストに追加してループを終える
- 最後のループでは x には"/e"が代入され，x は"/"で始まる文字列であり，x は X の最終要素であるため命令リストには"get e"が追加される
- 最終的には命令リストは{"go /", "go a//b", "test @c='d'", "get e" }となる

4.2.2 方法 3 の命令生成法

方式 3 では共通モジュール側で XPath 式をステップに分解し，それらを方式 3 のラッパーが提供するインターフェイスの命令に変換する必要があるが，その変換法を表 4.2 に

```
XPath 式を predicate で分割しリスト X にそれぞれを加える;
比較演算子を "=","!=","<","<=",">",">=" の文字列のどれか 1 つであると定義;
空のリスト "命令リスト" を作成;
"go /"を命令リストに追加;
for(i=0; i < X の要素数; i++) {
    x = X の要素 [i];
    if(x が"/"から始まる){
        x を "/" "相対ステップ" に分割;
        if(x が X の最終要素である){
            命令 "get 相対ステップ" を命令リストに追加;
        } else {
            命令 "go 相対ステップ" を命令リストに追加;
        }
    } else (x が"["から始まる){
        x を "[" "predicate 文字列" "]" に分割;
        if(x が X の最終要素である){
            命令 "get predicate 文字列" を命令リストに追加;
        } else {
            命令 "test predicate 文字列" を命令リストに追加;
        }
    }
}
```

表 4.1: 方式 2 の XPath を命令へ返還する疑似プログラム

疑似プログラムで示す.

表 4.2 の疑似プログラムを用いて /a//b[@c='d']/e を命令列に変換する例を以下に挙げる,

- リスト X は{ "/a","/", "/b","[@c='d']","/e" }となる．真偽値 f_descendant_or_self は偽である．x は X の最終要素では無いため 命令リストには "go a" が追加される.
- 次のループでは x には"/"が代入され, x は"/"から始まる文字列であり, f_descendant_or_self は偽であるが，x を "/" "ステップ名"に分割すると "ステップ名" は空文字列""であるため f_descendant_or_self を真にしてループを終える.
- 次のループでは x には"/b"が代入され, x は"/"から始まる文字列であり, f_descendant_or_self が真であり，x は X の最終要素では無いため 命令リストには "getAnyChild b"が追加され，f_descendant_or_self を偽にする.
- 次のループでは x には"[@c='d']"が代入され，x は"["から始まる文字列のため predicate 部であることがわかる．x には比較演算子が含まれるため "test @c"を命令リストに追加してループを終える.
- 最後のループでは x には"/e"が代入され, x は"/"で始まる文字列であり, f_descendant_or_self は偽であり，x は X の最終要素であるため命令リストには"get e"が追加される.
- 最終的には命令リストは{"go a","goAnyChild b","test @c","get e" }となる.

```
XPath 式をステップで分割しリスト X にそれぞれを加える;
比較演算子を "=","!=","<","<=",">",">=" の文字列のどれか 1 つであると定義;
真偽値 f_descendant_or_self を 偽にセットし, 空のリスト "命令リスト" を作成;
for(i=0; i < X の要素数; i++) {
    x = X の要素 [i];
    if(x が"/"から始まる){
        x を "/" "ステップ名" に分割;
        if(ステップ名が空文字列 "" である){
            x は"//"のため, f_descendant_or_self を真にする;
        } else {
            if(f_descendant_or_self が真){
                if(x が X の最終要素){
                    命令 "getAnyChild ステップ名" を命令リストに追加;
                } else {
                    命令 "goAnyChild ステップ名" を命令リストに追加;
                }
                f_descendant_or_self に偽をセット;
            } else {
                if(x が X の最終要素){
                    命令 "get ステップ名" を命令リストに追加;
                } else {
                    命令 "go ステップ名" を命令リストに追加;
                }
            }
        }
    } else (x が"["から始まる){
        x を "[" "predicate 文字列" "]" に分割;
        if("predicate 文字列"に比較演算子を含む){
            "Predicat 文字列" を "名前" "比較演算子" "値" に分割;
            命令 "get 名前" を命令リストに追加;
        } else {
            命令 "test predicate 文字列" を命令リストに追加;
        }
        if(x が X の最終要素である){
            命令 "get ."を命令リストに追加;
        }
    }
}
```

表 4.2: 方式 3 の XPath を命令リストへ返還する疑似プログラム

```
"命令" = "命令リスト"からラッパーに渡す命令を取得;
"ステップ名" = "命令" からステップ名を取得;
"regist ステップ名"をラッパーに渡す;
qd = ラッパーからクエリディスクリプタ qd を取得;
"命令"のステップ名を qd に置き換える;
"命令"をラッパーに渡す;
result = ラッパーからの返り値;
if(result が 真 もしくは偽以外の文字列){
  次の命令の処理を行う．次の命令が偽になると処理が返ってくる;
  "命令"が
  go qd の場合 goNext qd に,
  test qd の場合 textNext qd に,
  get qd の場合 getNext qd に置き換える;
} else if (result が 偽){
  処理を終了し，元のプログラムに戻る;
}
"命令"をラッパーに渡す;
result = ラッパーからの返り値;
while(真){
  if(result が 真 もしくは偽以外の文字列){
    次の命令の処理を行う．次の命令が偽になると処理が返ってくる;
    "命令"をラッパーに渡す;
    result = ラッパーからの返り値;
  } else if (result が 偽){
    処理を終了し，元のプログラムに戻る;
  }
}
```

表 4.3: 方式 3 の命令リストをラッパーに渡す時の疑似プログラム

4.3 各方式での共通モジュールとラッパーの協調動作

この節では，各方式での共通モジュールとラッパーとの協調動作について説明する．

4.3.1 方式 1 の共通モジュールとラッパーの協調動作

共通モジュールはラッパーを起動し，XPath 式と MBX ファイル名をセットするとラッパー側で XPath 式の評価がされ，その結果を共通モジュール側に通知しラッパーは終了する．共通モジュール側はラッパーから受け取ったデータをユーザに提示して終了する．

4.3.2 方式 2 の共通モジュールとラッパーの協調動作

共通モジュールはラッパーに対し命令を送り，その結果に基づいて共通モジュールは動作を決定する．共通モジュール側はラッパーから返された値が真もしくは位置情報や文字列の場合，リストの次の命令の登録命令をまずをラッパーに渡し，返されたクエリディスクリプタを次の命令の相対パスに置き換えてからラッパーに渡す．偽の場合はリストの一つ前の命令に戻り，移動命令の場合は go next 命令を発し，テスト命令の場合は test next 命令を発し，取得命令の場合は get next 命令を発する．このときのクエリディスクリプタは偽と判別されたクエリディスクリプタの一つ前のクエリディスクリプタを使用する．リストの先頭の命令に対しラッパーが偽を返すと，動作を終える．この動作は表 4.3 に示している．また，次に動作例を挙げる．

/x/y[3]/text() という XPath 式の場合，リストには { go /, go x/y，test 3，get text() } が格納され，リストの最初から順に登録命令と命令がラッパーに渡される．test 3 の時点で 3 番目の y ノードがデータ中に存在する場合は次命令の text() の登録命令をラッパーに渡し，そのようなノードが存在しない場合はラッパーから false が返るため，一つ戻り go next x/y の x/y を 1 つ前のクエリディスクリプタ qd1 に置き換えた命令をラッパーに渡し，最終的に go next qd1 が偽になると終了する．

また，/x/y[@index=3]/text() という XPath 式の場合はリストには {go /, go x/y，test @index=3，get text() } が格納される．test @index により ラッパーは非 XML データの/x/y ノードの attribute である index の値を取得する．取得した値はラッパー内で 3 と比較され，同値なら次の get text() を発行し，違うなら go next x/y 命令の x/y を 1 つ前のクエリディスクリプタ qd1 に置き換えた命令をを発効して次の/x/y ノードの index 値のテストをラッパーに指示する．この処理は go next qd1 命令の結果が false となるまで繰り返される．

/a/b//c という XPath 式の場合，リストには {go /, get a/b//c} という命令が格納され先頭から処理される．get /a/b//c は最初に出された後，その結果が true なら false が返るまで a/b//c を 1 つ前のクエリディスクリプタ qd0 に置き換えた get next ad0 の発行を繰り返す．

4.3.3 方式 3 の共通モジュールとラッパーの協調動作

共通モジュールはラッパーに対し命令を送り，その結果に基づいて共通モジュールは動作を決定する．共通モジュール側はラッパーから返された値が真の場合，リストの次の命令をラッパーに渡す．偽の場合はリストの一つ前の命令に戻り，移動命令の場合は goNext 命令を発し，テスト命令の場合は testNext 命令を発し，取得命令の場合は getNext 命令を発する．リストの先頭の命令に対しラッパーが偽を返すと，動作を終える．動作例を次に挙げる．

/x/y[3]/text() という XPath 式の場合，リストには go x，go y 3，get text() が格納され，リストの最初から順に命令がラッパーに渡される．go y 3 の時点で 3 番目の y ノードがデータ中に存在する場合は次の text() をラッパーに渡し，そのようなノードが存在しない場合はラッパーから false が返るため，この時点で XPath 式の評価を終える．

また，/x/y[@index=3]/text() という XPath 式の場合はリストには go x，go y，get @index，get text() が格納される．get @index により ラッパーは非 XML データの/x/y ノードの attribute である index の値を取得する．取得した値は共通モジュールで 3 と比較され，同値なら次の text() を発行し，違うなら goNext y 命令を発効して次の/x/y ノードの index 値の取得をラッパーに指示する．この処理は goNext y 命令の結果が false となるまで繰り返される．

/a/b//c という XPath 式の場合，リストには go x，go y，getAnyChild c という命令が格納され先頭から処理される．getAnyChild c は，/a/b の子孫である最初の c ノードに出会うと一度 true を返すが，共通モジュールはここでさらに getAnyChild c 命令を出し，これをラッパーから false が返るまで繰り返す．

4.3.4 方式 4 の共通モジュールとラッパーの協調動作

共通モジュールはラッパーをインスタンスとして作成し，XSQ の SAX エンジンとして使用する．XSQ からは結果が提示されるため，その結果をユーザに提示して終了する．

第5章　システムの実装

実装として方式1〜4の共通モジュール及びラッパーを Java 5.0 で実装した．また，ラッパーの例として，メールボックス形式の一つである MBX 形式を対象としたラッパーを実装した．

5.1　各方式がサポートする XPath 命令群

この節では，各方式がサポートする XPath 命令群を説明する．

5.1.1　方式 1 がサポートする XPath 命令群

方式 1 でサポートする XPath 命令は全ての XPath 命令である．XPath 命令は上記で示したように共通モジュール側では処理されず，ラッパーで処理される．

5.1.2　方式 2 がサポートする XPath 命令群

方式 2 でサポートする XPath 命令は以下の通りである．

- **axis**: XPath の axis としてよく使用される child，attribute，parent，descendant-or-self をサポートする．これらの XPath 式の axis とその省略形は表 2.4 に示されている．
- **NodeTest**: XPath 式のノードテストのうち，要素名によるノードテスト及び text() をサポートする．
- **predicate**: 方法 2 では XPath の predicate のうち，[80] や [@x=”z”]，[x/y/z/@a=”b”] といった，数字のみ，値を調べるもののみ，相対ロケーションパス，相対ロケーションパスと値を調べるものを複数合成した predicate をサポートする．入れ子構造になっている predicate はサポートしない．

5.1.3　方式 3 がサポートする XPath 命令群

方式 3 でサポートする XPath 命令は以下の通りである．

- **axis**: XPath の axis としてよく使用される child，attribute，parent，descendant-or-self をサポートする．これらの XPath 式の axis とラッパーが提供する移動命令との変換表を表 2.4 に示す．
- **NodeTest**: XPath 式のノードテストのうち，要素名によるノードテスト及び text() をサポートする．
- **predicate**: 本研究では XPath の predicate のうち，[80] や [@x=”y”]，[x] といった，数字のみ，値を調べるもののみ，ノードテストのみの predicate をサポートする．入れ子構造になっている predicate はサポートしない．

5.1.4 方式 4 がサポートする XPath 命令群

方式 1 でサポートする XPath 命令は全ての XPath 命令である．

5.2 MBX 形式

MBX 形式のファイルには一つ以上の電子メールが保存され，一つの電子メールは「From -」で始まり，その後に RFC822 で定められたフォーマットに基づきデータが配置され，一回の空行と「From -」で始まる文字列までである．メールの構造を式に直したものを表 5.1 に示す．ここで，#STRING とは制御文字以外の ISO-8859-1 の文字セットである．

また，本研究では大量のメールデータが必要となるため，メールデータ生成プログラムを作成した．1 メールにつきヘッダ数 3，本文部 1，本文部の長さは 0 から 5000 単語までのランダムの数の単語が辞書データから生成される．辞書データは `ftp://ftp.ox.ac.uk/pub/wordlists/` にある辞書データを使用した．このソフトにより生成されたメールデータを表 5.2 に示す．また，後述するが方法 1 では MBX データではなく XML データを用いる為，MBX データから XML データに変換するプログラムも作成した．表 5.2 のメールデータを XML 形式に変換したものを表 5.3 に示す (実際の XML データは<mail>が一番最上段の要素ではなく，さらに<mail>群を包含する<mbx>要素が存在します)．

5.3 共通モジュールの実装

この節では実際に実装した各方式の共通モジュールについて説明する．実際にどのようにラッパーと協調して動作するかは，節 4.3 で示す．

表 5.1: MBX 構造の文法

MBX	:=	(MAIL)*
MAIL	:=	HEADERS, BODY
HEADERS	:=	(HEADER)+
HEADER	:=	HEADERNAME ":" HEADERVALUE
HEADERNAME	:=	#STRING
HEADERVALUE	:=	#STRING
BODY	:=	#STRING

表 5.2: メールデータの例

From -
To : hoge@jaist.ac.jp
From : syphilis@jaist.ac.jp
Subject : foxtrot

liveandletlive bitching edges trapdoor dialin margaux aztecs
digital backrub ringo

password thinthighs webetoys badtimes turnleft whereisthebeef
ibmsux fastbreak jellystone spam fubar network
zener blowjob chardonnay
sombrero breastfeed uptohere spiffy makingit
berlinwall tsing tao
boobys bigboy cabinboy
dead-head c++ newuser fuzzbat
slimeball merrychristmas takeiteasy batmobile

表 5.3: メールデータを XML に変換した例

```
<mail>
  <headers>
    <header name="To" value="hoge@jaist.ac.jp" />
    <header name="From" value="syphilis@jaist.ac.jp" />
    <header name="Subject" value="foxtrot" />
  </headers>
  <body>
liveandletlive bitching edges trapdoor dialin margaux aztecs
digital backrub ringo

password thinthighs webetoys badtimes turnleft whereisthebeef
ibmsux fastbreak jellystone spam fubar network
zener blowjob chardonnay
sombrero breastfeed uptohere spiffy makingit
berlinwall tsing tao
boobys bigboy cabinboy
dead-head c++ newuser fuzzbat
slimeball merrychristmas takeiteasy batmobile
  </body>
</mail>
```

5.3.1 方式 1 の共通モジュールの実装

方式 1 で用いる共通モジュールは起動するラッパーの種類として MBX 形式をサポートするラッパーを起動し，起動パラメータとして MBX ファイル名と XPath 式を設定する．その後，ラッパーからの入力を待ち受け，ラッパーからの入力イベントが発生するとラッパーからの入力データをユーザに提示する．後述する方式 2,3 とは違いラッパーからの入力に基づく動作の変更は無い．

5.3.2 方式 2 の共通モジュールの実装

方式 2 で用いる共通モジュールは，ユーザから XPath 式と MBX ファイル名を受け取り，起動するラッパーの種類として MBX 形式をサポートするラッパーを起動し，起動パラメータとして MBX ファイル名を設定する．

次に，節 4.2.1 に示した疑似プログラムに基づき XPath 式を命令リストに変換し，このリストに基づいてラッパーに対して順に命令を送るが，それらの命令の前に登録命令をラッパーに送り，ラッパーからはクエリディスクリプタと呼ばれる識別子を取得して次の命令時に simple path をクエリディスクリプタに置き換える．ラッパーから位置情報，真偽値，その他文字列が返るのでその値に基づき次の動作を決定する．最終的に命令リストの命令が全て真になるか，先頭の命令が偽であると判断されると動作を終了し，ユーザに結果を提示する．

5.3.3 方式 3 の共通モジュールの実装

方式 3 で用いる共通モジュールは，ユーザから XPath 式と MBX ファイル名を受け取り，起動するラッパーの種類として MBX 形式をサポートするラッパーを起動し，起動パラメータとして MBX ファイル名を設定する．

次に、節 4.2.2 に示した疑似プログラムに基づき XPath 式を命令リストに変換し，このリストに基づいてラッパーに対して順に命令を送る．ラッパーからは真，偽，及びその他文字列が返るので，その値に基づき次の動作を決定する．最終的に命令リストの命令が全て真になるか，先頭の命令が偽であると判断されると動作を終了し，ユーザに結果を提示する．

5.3.4 方式 4 の共通モジュールの実装

方式 4 では MBX ファイルへの XPath 式の適用に SAX インターフェイスを用いるため，共通モジュールとラッパーはそれぞれ違うクラスとして実装し，共通モジュールからラッパーのインスタンスを作成し，そのインスタンスに対しユーザから受け取る MBX ファイルを設定する．ラッパーからは SAX イベント列が返されるので，その SAX イベントに対

応する SAX イベントハンドラを用いて XPath 式の評価を行う．この SAX イベント列を用いて XPath 式を評価するシステムとして，既存の技術である XSQ を改造して用いた．XSQ が使用する SAX エンジンにラッパーを指定することで，XML データでない MBX ファイルであっても XSQ を使用することができる．そして，XSQ が XPath 式を評価した結果をユーザに提示する．

5.4 ラッパーの実装

この節では各方式でのラッパーの実装について説明する．

5.4.1 方式 1 用ラッパーの実装

方式 1 用ラッパーは共通モジュールから起動され，XPath 式と MBX データファイルを受け取る．その後 MBX データファイルに対して，XPath 式を適用することで結果を取得する．本研究では，MBX データファイルに対しての XPath 式の適用に，XML ファイルに対し XPath 式を適用することができる Apache プロジェクトの Xalan を使用した．Xalan はデータとして XML データを要求するため，実験時には先に MBX ファイルを XML データに変換するツールを作成して MBX ファイルに使用し，その結果できた XML データを用いることとする．そして，Xalan の XPath 式の処理結果を共通モジュールに渡す．

5.4.2 方式 2 用ラッパーの実装

ラッパーは共通モジュールから起動され，共通モジュールからの停止命令により終了する．MBX データ用のラッパーは，共通モジュールからの命令により MBX データ上でファイルポインタを移動し，成功するとファイルポインタもしくは真偽値，あるいは取得データを返す．内部処理のアルゴリズムを表 5.4 に示す．

MBX データを木構造で表した図が図 5.1 である．ここで各ノードの左上の数字はノード番号であり，右上の数字はファイルポインタの位置である．また，ノード番号 4 のテキストノードの値は”fail”で，ノード番号 8 のテキストノードの値は”success”であるとする．このとき，共通モジュールに XPath 式 /mbx/mail[headers/header='success']/body/text() が与えられたときのラッパーの処理を以下に示す．

1. ラッパーは ”regist / / 0”命令が渡されるとこの命令を命令リストに追加し，クエリディスクリプタ qd1 を共通モジュールに渡す．

2. 次に ”go qd1”を受け取りると，qd1 に対応する移動先”/”を命令リストから取得し，”/”に移動し，位置情報 0 を返す．

3. 次に ”regist mbx/mail / 0”命令を受けとると，この命令を命令リストに追加し，クエリディスクリプタ qd2 を共通モジュールに渡す.

4. 次に ”go qd2”を受け取ると，qd2 に対応する移動先”mbx/mail”を命令リストから取得し，最初の”mbx/mail”に移動し，位置情報 0 を返す.

5. 次に ”regist headers/header=’success’ mbx/mail 0” を受け取ると，この命令を命令リストに追加し，クエリディスクリプタ qd3 を共通モジュールに渡す.

6. 次に ”test qd3”を受け取ると，qd3 から命令リストから対応するテスト ”headers/header=’success’”を取得する．その後現在位置から headers/header の値を取得し，’success’ と同値であるかを比較する．今回はノード番号 4 の header 要素のテキストノードの値が”fail”であり，その他に”headers/header”に対応する要素がないため，qd3 から位置情報を読み取り，一つ前の移動命令”go qd2”によって返された位置情報である 0 に移動し，命令リストから qd3 削除し共通モジュールに false を渡す.

7. 次に ”next qd2”を受け取ると，ノード 2 より後ろにある”mbx/mail”に移動し，位置情報 100 を共通モジュールに渡す.

8. 次に ”regist headers/header=’success’ mbx/mail 100” を受け取ると，この命令を命令リストに追加し，クエリディスクリプタ qd3 を共通モジュールに渡す.

9. 次に ”test qd3”を受け取ると，qd3 から命令リストから対応するテスト ”headers/header=’success’”を取得する．その後現在位置から headers/header の値を取得し，’success’ と同値であるかを比較する．今回はノード番号 8 の header 要素のテキストノードの値が”success”であり共通モジュールに true を渡す.

10. 次に ”regist body/text() mbx/mail 100”を受け取ると，この命令を命令リストに追加し，クエリディスクリプタ qd4 を共通モジュールに渡す.

11. 次に ”get qd4”を受け取ると，body 要素のテキスト値を取得し，共通モジュールにその文字列を渡す.

```
空のスタックである命令スタック を作成;
int counter = 0; //クエリディスクリプタ用のカウンタ
while(真){
  命令 = 共通モジュールからの命令を読み込む;
  真偽値 result = 偽;
  switch(命令){
  case 登録命令;
    result = regist(命令);
    break;
  case 移動系命令:
    result = go(命令);
    break;
  case テスト系命令:
    result = test(命令);
    break;
  case 取得系命令:
    result = get(命令);
    break;
  }
  if(result が真で, 命令がテスト系命令である){
    共通モジュールに true を入力;
  } else {
    共通モジュールに false を入力;
  }
}
```

表 5.4: 方式 2 ラッパー内部の疑似プログラム

```
真偽値 regist(命令){
  命令スタックに命令を追加して counter ++;;
  共通モジュールにクエリディスクリプタ ("qd"+counter) を渡す;
}

真偽値 go(命令){
  fp = 現在のファイルポインタ;
  index = 命令からクエリディスクリプタを抜き出し，その番号を取得;
  path = 命令リスト [index] の 相対パス x を取得;
  真偽値 result = path に移動を行い，移動に成功した場合真，失敗したら偽;
  //このとき，result が真なら現在のファイルポインタの位置が移動
  if(result が真){
    共通モジュールに fp の値を渡す;
  } else {
    現在のファイルポインタを fp にセット;
  }
  return result;
}
```

表 5.5: 方式 2 ラッパー内部の疑似プログラムのサブルーチン群 1

```
真偽値 test(命令){
  index = 命令からクエリディスクリプタを抜き出し，その番号を取得;
  path = 命令リスト [index] の 相対パス x を取得;
  真偽値 result = false;
  if(path に比較演算子が入っている){
    result = path に移動し，その値を取得し，path の比較対象と比較を行った結果;
  } else {
    result = path に移動できるかどうか;
  }
  // result の値にかかわらず現在のファイルポインタの位置は変動せず
  return result;
}

真偽値 get(命令){
  index = 命令からクエリディスクリプタを抜き出し，その番号を取得;
  path = 命令リスト [index] の 相対パス x を取得;
  真偽値 result = path の値の取得に成功した場合真，失敗したら偽;
  // result の値にかかわらず現在のファイルポインタの位置は変動せず
  if(result が真){
    共通モジュールに値を渡す;
  }
  return result;
}
```

表 5.6: 方式 2 ラッパー内部の疑似プログラムのサブルーチン群 2

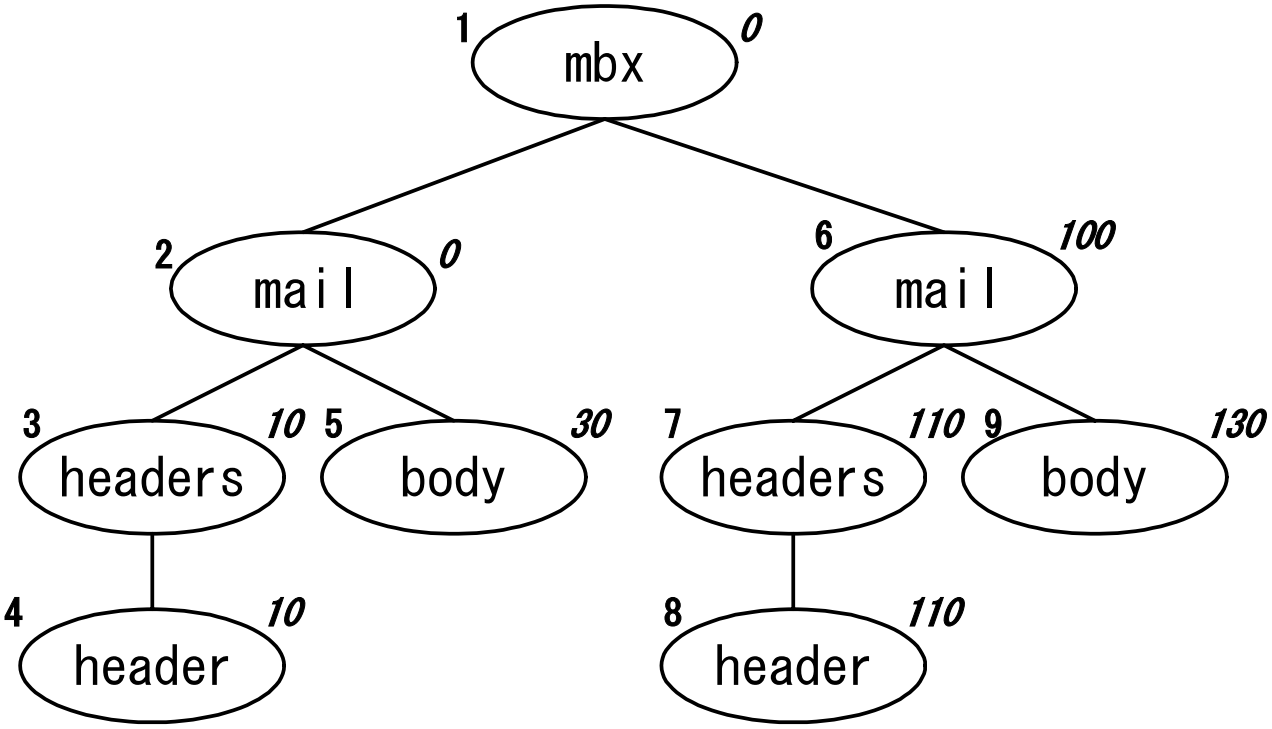

図 5.1: MBX データの木構造での表現

5.4.3 方式 3 用ラッパーの実装

ラッパーは共通モジュールから起動され，共通モジュールからの停止命令により終了する．MBX データ用のラッパーは，共通モジュールからの命令により MBX データ上でファイルポインタを移動し，成功すると true もしくは取得データを返し，失敗すると false を返す．追加の処理として内部ではファイルの先頭から現在読み込んだ位置までに go 命令で移動したノードの XPath 式による階層表現とファイルポインタのオフセットをセットで保持する。また，共通モジュールから渡された命令が取得命令の場合，その値を共通モジュールに渡す．内部処理のアルゴリズムを表 5.7 に示す．

MBX データを木構造で表した図が図 5.1 である．ここで各ノードの左上の数字はノード番号であり，右上の数字はファイルポインタの位置である．共通モジュールに XPath 式 /mbx/mail[2]/body/text() が与えられたときのラッパーの処理を以下に示す．

1. ラッパーは go mbx を受け取るとラッパー内部のアルゴリズムに則り，命令が移動系命令のため go() に入る．ルートノード以下に要素 mbx があるため，result は真となるのでノード番号 1 の要素 mbx に移動し，スタックに ”/mbx 0” を追加する

2. 次に go mail を受け取るとラッパー内部のアルゴリズムに則り，命令が移動系命令のため go() に入る．現在のノード以下に mail が存在ノード番号 2 の要素 mail に移動し，スタックに ”/mbx/mail 0” を追加する

3. 次に goNext mail 1 を受け取ると 1 つ次の兄弟要素 mail に移動し，スタックに”/mbx/mail 100”を追加する

4. 次に go body を受け取ると 1 つめの body 要素に移動し，スタックに ”/mbx/mail/body 130” を追加する

5. 最後に get text() を受け取るとノード番号 9 のテキスト値を取得して共通モジュールに渡す

5.4.4 方式 4 用ラッパーの実装

ラッパーは共通モジュールからインスタンスとして作成され，MBX データファイルを受け取り，SAX イベントを発生させる．MBX データファイルは XML データでは無いため，先頭から順にファイル内を走査し，仮想的に MBX データファイルが XML データであるかのように SAX イベントを発生させる簡易 SAX エンジンを作成した．

```
空のスタックである命令スタック を作成;
while(真){
  命令 = 共通モジュールからの命令を読み込む;
  真偽値 result = 偽;
  switch(命令){
  case 移動系命令:
    result = go(命令);
    break;
  case テスト系命令:
    result = test(命令);
    break;
  case 取得系命令:
    result = get(命令);
    break;
  }
  if(result が真){
    共通モジュールに true を入力;
  } else {
    共通モジュールに false を入力;
  }
}
```

表 5.7: 方式 3 ラッパー内部の疑似プログラム

```
真偽値 go(命令){
  fp = 現在のファイルポインタ;
  真偽値 result = 命令の種別に従って処理を行い移動に成功した場合真，失敗したら偽;
  //このとき，result が真なら現在のファイルポインタの位置が移動
  //この時の命令が 親に移動する命令の場合，スタックから 2 回 pop を行い，2 回目の pop
  //には親のファイルポインタが格納されているため，その情報を用い場所移動を行う
  if(result が真){
    ノードの XPath 表現と現在のファイルポインタのセットを命令スタックに追加;
  } else {
    現在のファイルポインタを fp にセット;
  }
  return result;
}

真偽値 test(命令){
  真偽値 result = 命令の種別に従って処理を行いテストに成功した場合真，失敗したら偽;
  // result の値にかかわらず現在のファイルポインタの位置は変動せず
  return result;
}

真偽値 get(命令){
  真偽値 result = 命令の種別に従って処理を行い，値の取得に成功した場合真，失敗したら偽;
  // result の値にかかわらず現在のファイルポインタの位置は変動せず
  if(result が真){
    共通モジュールに値を渡す;
  }
  return result;
}
```

表 5.8: 方式 3 ラッパー内部の疑似プログラムのサブルーチン群

第6章　実験及び評価

非XMLデータとして電子メールボックス形式の一つであるMBX形式を用いた評価実験を行った．実験環境は，PC(Pentium M 1.4GHz，RAM 512MB，Windows XP SP2，J2SDK 5.0)上である．実験には格納されているメール数がそれぞれ違うMBXファイルを5つ用い，それぞれ格納されているメール数は1000, 2000, 3000, 4000, 5000である．それぞれのメールデータは節5.2で示したメール作成ソフトを使用してメールデータを生成した．

6.1　実験方式

上記の条件で，XPath式/mbx/mail[n]/body/text()を用いて各方式の実験を行った．ここで，nは0からそれぞれの格納データ数までの，500毎の値である．

6.2　実験結果

方式1においてMBXファイルをXMLデータに変換した後に，XMLデータに対し上記のXPath式を用いて問い合わせした実行時間を図6.1に示す．

方式3のMBXファイルに対し上記のXPath式を用いて問い合わせた実行時間を図6.2に示す．

方式4のMBXファイルに対し上記のXPath式を用いて問い合わせた実行時間を図6.3に示す．

6.3　考察

方式1,3,4の実行時間を図6.4に示す．方式1,3,4のメモリ使用量を表6.5に示す．それぞれ3回データをとり，その平均を採用した．また，ここでのメモリ使用量とは，共通モジュールとラッパーのメモリ使用量を合計したものである．

図6.4の各方式を比較すると本研究で使用したシステムの方が高速な結果となり，メモリ使用量も方式1,4と比べて少なくなることがわかった．しかし，方式4が方式1の処理結果より遅くなってしまった．これは，どちらのラッパーもMBXファイルもしくはXMLファイルを1パスで処理することを考えると，方式1のラッパーがオープンソースのプログラムを使用していることに対し，方式4のラッパーが自作のプログラムを使用すること

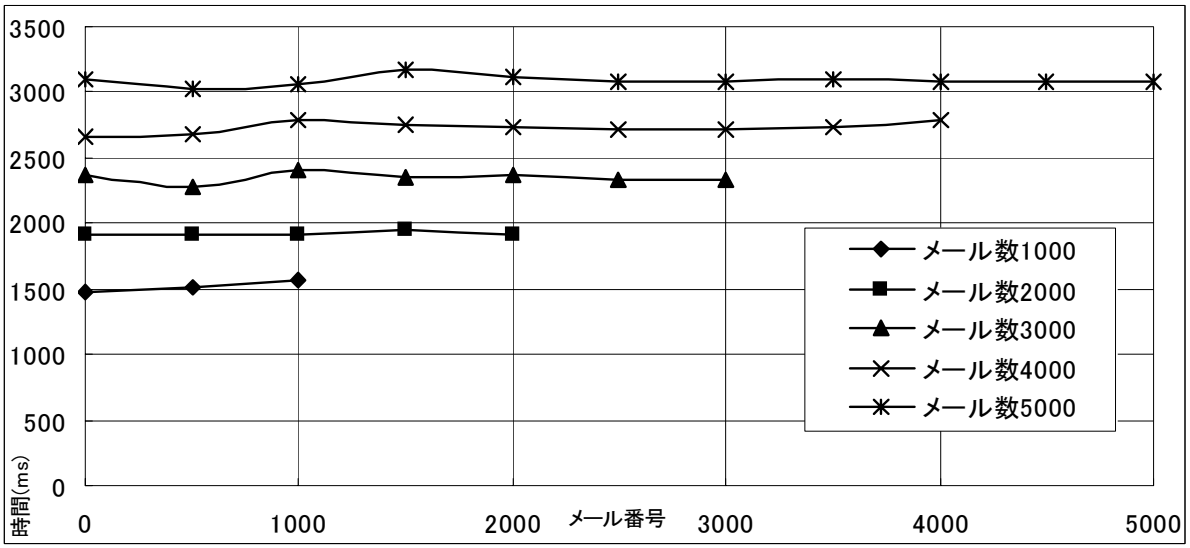

図 6.1: 方式 1 の処理時間

で，速度を重視したチューニングができていない可能性がある．また，方式 2 が一番早くなるの理由は，方式 1 や 4 は図 6.4 によると，総メール数が大きくなるほど処理時間は大きく，同じ方式で，同じ総メール数の場合の XPath 式の処理時間は変わらない．これは，方式 1 や 4 がデータを一度最後まで読まなければならないためである．しかし，方式 2 では/mbx/mail[n +1] 以降のノードは処理する必要がないため，その時点で処理を打ち切ることができる．よって，n の値が低い場合は高速な処理を行うことができる．また，方式 1 の結果は MBX から XML への変換時間を含まない．この変換時間を足すと，非 XML データに対して XPath による問い合わせを行う場合，方式 1 を使用するよりも当システムを使用する方が処理時間及びメモリ使用量に関して有効であることがわかる．方式 1,3,4 のそれぞれのコード量は，方式 1 が一番少なく，次に 4,3 となった．この理由は，方式 1 は共通モジュールの機能自体が少なく，ラッパーも既に存在しているライブラリを使用することができたためであり，方式 4 も XPath 式の処理に既存のライブラリを使用することができたからだ．しかし，方式 3 は共通モジュールやラッパーなどを全て 1 から作成したため，コード量が多くなってしまった．

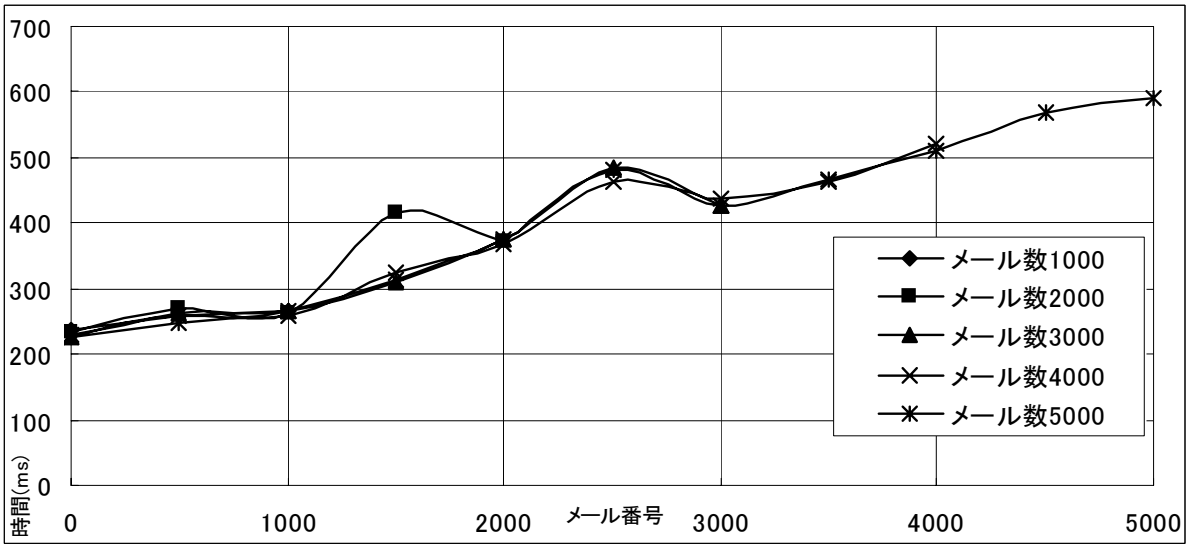

図 6.2: 方式 3 の処理時間

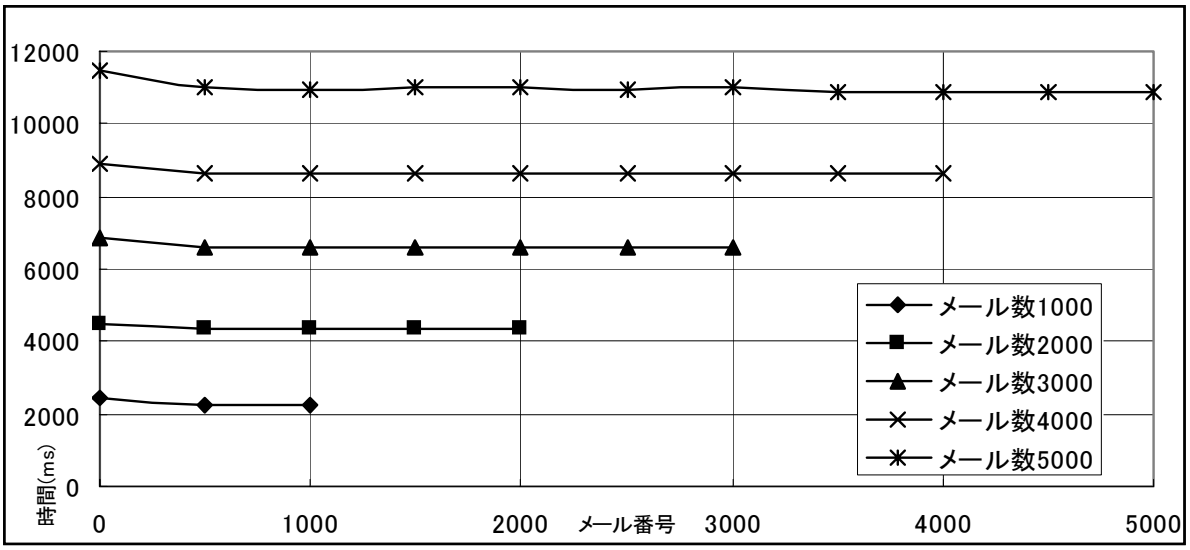

図 6.3: 方式 4 の処理時間

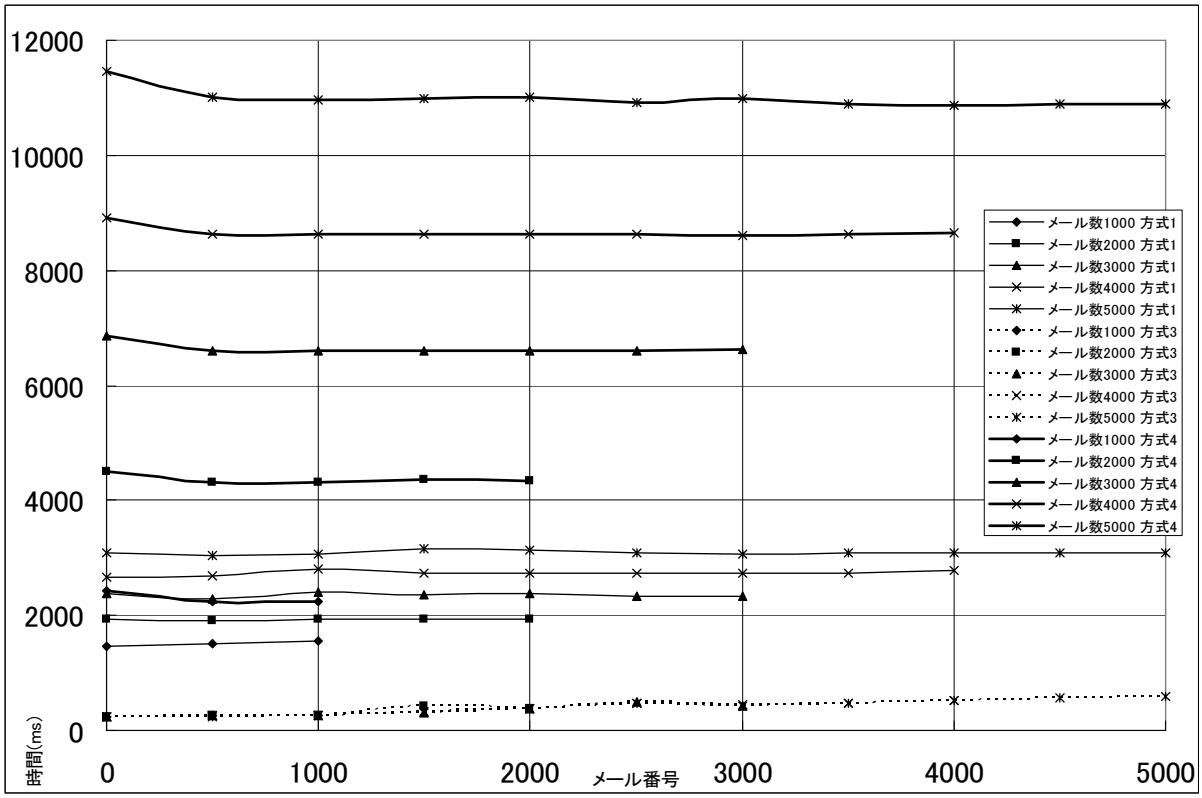

図 6.4: 方式 1,3,4 の処理時間

メール総数	方式 1 (K byte)	方式 3 (K byte)	方式 4 (K byte)
1000	21442	1402	5004
2000	44250	1478	5004
3000	64213	1956	5005
4000	87598	2125	5005
5000	106109	2294	5006

図 6.5: 方式 1,3,4 のメモリ使用量

第7章　まとめと今後の課題

本研究では非 XML データに対する XPath 検索のためのラッパーの提供するインターフェイスのデザインについて考察し，有効と考えられる方式の一つについて実装と評価実験を行った．今回は電子メールボックス形式の一つである MBX について評価を行ったが，今後，他形式の非 XML データについても実験を行う予定である．また，現時点では方式 2 の実装が終わっていないが，実装を終わらせて方式 3 との比較実験を行ったり，その他のインターフェイスを考案して実験を行い，最適なインターフェイスを探りたいと考えている．

謝辞

本研究を行うに当たり，ご指導・ご鞭撻を頂いた田島敬史助教授に心から感謝し，深くお礼申し上げます．また，日常，有益な議論をして頂いた研究室の皆様にも感謝します．

参考文献

[1] eDIKT::BinX. http://www.edikt.org/binx.

[2] Xalan. http://xml.apache.org/xalan-j/index.html.

[3] J. Clark and S. DeRose, eds. *XML Path Language (XPath) Version 1.0 – W3C Recommendation*. http://www.w3.org/TR/xpath, Nov. 1999.

[4] 品川徳秀 and 北川博之. バイナリデータ上の XML ビュー機構と XPath 処理の提案. *DBSJ Letters*, 2(3):49–52, Dec. 2003.

[5] RELAX NG. http://www.relaxng.org/spec-20011203.html.

[6] jaxen. http://jaxen.org/.

[7] XSQ. http://www.cs.umd.edu/projects/xsq/