構造化文書の応需計算機構*

By-Need Evaluation of Programmable Structured Documents

西岡 真吾 † Shingo NISHIOKA
中野 圭介 † Keisuke NAKANO
胡 振江 †‡ Zhenjiang HU
武市 正人 † Masato TAKEICHI
† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, University of Tokyo
‡ 科学技術振興機構 さきがけ研究 21
PRESTO 21, Japan Science and Technology Agency
{nis,ksk,hu,takeichi}@mist.i.u-tokyo.ac.jp

PSD (Programmable Structured Documents) は構造化文書を計算可能なプログラムとして捉えることで安全な文書処理を実現する機構である．その計算の実行 (評価) は文書を保持するエディタ等のプロセスとは別の外部評価系により行われるため, 評価される文書をプロセス間で授受する効率的な手段が必要となる．応需 DOM は外部評価系が他のプロセスが保持している文書へ効率的にアクセスするための汎用の手段を与える．応需 DOM では, 外部評価系で文書の評価が進行し文書の一部が必要となる都度その部分のみを転送するため, 実際に転送される文書量は最小限になる．文書の部分の指定には XPath のサブセットが使用されるため, 文書データを DOM 等により保持するサーバとの親和性は高く実装も容易である．また, 応需 DOM では PSD の外部評価系で必要となる必要最小限のアクセスメソッドのみを実装しており, シンプルであるためその移植性は高い．

1 はじめに

電子的な構造化文書情報の蓄積と効果的な情報利用技術は, インターネットを含む広範な情報の交換・流通にとってきわめて重要な位置を占めている．XML に代表されるこれらの技術は, 発展の著しい Web による情報環境に向けて既存の技術の延長線上で実務的に開発されたものであり, 事実上の標準となってはいるがその言語的な概念が十分には整理されていない．このような体系的な処理技術の欠如が今後の情報交換の発展を阻害し, 既存技術による個別的対処や人手による個別対応では, 一般性を欠く文書情報を蓄積することとなっており, この問題を解決することが重要な課題となっている．このような構造化文書はプログラミング言語のデータ構造と類似しており, 関数型言語によるアルゴリズム記述が文書処理に適している．そこで, プログラムを手続き的に扱うソフトウェアによって文書の信頼性を高め, 情報流通を高度化することが可能となる．この問題に対処すべく, Programmable Structured Document(PSD) は, プログラムの記述を含む文書を対象として, 構造化文書処理を効果的に実現しようとするものである．すなわち, PSD は構造化文書をプログラミングにおける構造化データであるとみなし, プログラミング言語に関する理論を適用することによって, 安全かつ信頼性の高い処理を実現する．また, 処理を行うコードは対象文書に埋め込まれており, これによって文書の高い可搬性を実現している．

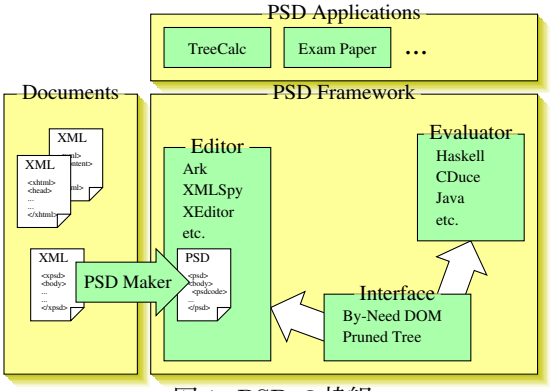

図 1: PSD の枠組

図 1 に PSD の枠組の全体構成を示す．PSD の枠組は既存の XML 文書を PSD に変換する PSD Maker, PSD の編集, 表示を行うエディタ, PSD の評価 (ここでは計算と同じ) を行う外部評価系, エディタと外

*本研究は, 文部科学省平成 15 年度リーディングプロジェクトである e-Society 基盤ソフトウェアの総合開発の一環として行われた「高信頼構造化文書変換技術」による．

部評価系と通信を媒介する通信インタフェース, そして個々の PSD アプリケーションから構成される. なお, PSD アプリケーションは処理対象となる構造化文書そのものとそれに埋め込まれた処理コード (処理プログラム) とから構成される. 本論文で述べる応需 DOM はこれらのうち通信インタフェースに相当する.

一般に, PSD の枠組においては文書がその文書自身を扱うことができるようなコードを含むが, そのコードを記述するプログラム言語は限定されないことが望まれるため, PSD を実行するためのプラットフォームとしては, 図 1 に示すように XML 文書を保持・処理する XML エディタ等と, PSD に埋め込まれたコードを実行する外部評価系とから構成される. このような構成の下では, 外部評価系からエディタが保持するデータに対して自由に参照・更新できる機構が必要になる. 例えば, PSD の代表的アプリケーションである TreeCalc[4] の場合には XML 文書としての TreeCalc を編集・表示する XML エディタとして Java で実装された XML 操作環境を用いる一方で, コードの評価系は CDuce[1] で記述されている. このため TreeCalc では Java が実行プロセス内にのみ保持している DOM に対して, CDuce から参照・更新するための機構が必要となる. 単純に DOM オブジェクト全体に対応する XML を生成して外部評価系に渡すという手法もあるが, 巨大なデータを扱う場合には, メモリ消費と実行時間の両面において効率が悪く, 特に, 埋込コードによる評価が部分的なデータにしか依存しない場合には, 著しく無駄の多い処理が行われることになる.

応需 DOM は, 外部評価系と XML エディタの間に介在し, 計算に必要となった部分のみを必要に応じて XML エディタより取得し, 外部評価系に渡すことにより, 不必要なデータのやりとりを回避するための基盤となる機構である. 応需 DOM 機構では通信時のオーバーヘッドが増えるが, 関数型言語等, 遅延評価機構を備えた汎用プログラミング言語と組み合わせることで, より効率的な評価が実現され, 全体としての実効的な効率を向上させることが可能となる.

なお, もうひとつの通信インタフェースの実現として Pruned-Tree[2] が現在実装されている. これは, コードの実行前に必要となる部分木を全て決定し, その部分木のみを外部評価系に転送するための汎用の枠組である. Pruned-Tree 方式は通信の断片化が発生しないため, オーバーヘッドを小さく抑えることが可能となる一方, ユーザレベルのプログラミングが難しく, 応需 DOM と比較しても一長一短であり, 使用場面に応じてこれらを使い分けることが重要となる.

2 応需 DOM の概要

応需 DOM は, XML データへの外部評価系からのアクセスを可能にするプロトコルである. 応需 DOM でのアクセス単位は要素 (属性および子孫を含まない) もしくは CData(テキスト) であり, 外部評価系がより大きな XML データの部分にアクセスするためには, 複数回に分けて (特に対象が部分木であれば再帰的に) アクセスする必要がある. 通信データは XML で表現され, XML との親和性が高いことも特徴のひとつである. 本節では応需 DOM の基本設計およびプロトコルについて説明する.

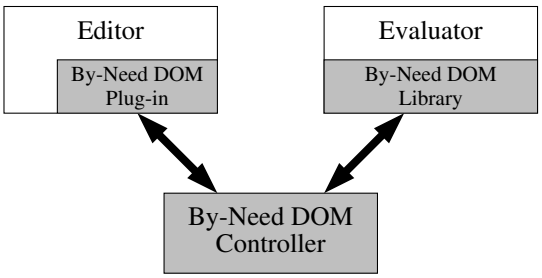

図 2: 応需 DOM の形態

図 2 でハッチングされている部分が応需 DOM である. 応需 DOM はエディタ側にはプラグインとして組み込まれ, そのためエディタは応需 DOM サーバとして機能する. 外部評価系側は応需 DOM ライブラリをリンクすることで応需 DOM クライアントとして動作する. このように PSD の枠組では応需 DOM はエディタへのプラグインおよびクライアント用ライブラリとから構成される.

2.1 応需 DOM のプロトコル

クライアント・サーバ間で用いられるプロトコルは XML ベースであり, クライアントによる要求とそれに対するサーバからの応答の対を最小単位とし, ひとつのセッションはその繰返から構成される. 要求は初期化を行う `init`, ノード取得要求 (`get`) とノード更新要求 (`put`), の 3 種類である. 図 3 に応需 DOM のプロトコルとデータフローの概略を示す.

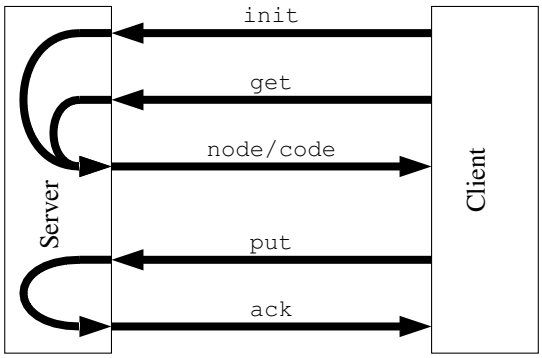

図 3: 応需 DOM のプロトコル

2.2 初期化: init

外部評価系は応需 DOM サーバに接続した後 init を送信する. その応答は, エディタ上で利用者が評価の実行を指定したノードのパスおよびそのノード以下の部分木である. PSD では, この部分木は実行すべきコードを含む. init で指定されたノードに含まれるコードを評価し, 結果をエディタに送り返すことが外部評価系の目的である.

init の応答ではエディタで評価するノード以下の部分木が全て送られる. 通常はこの木全体が必要となるため, クライアントがそれを再帰的に取得する負担を減らすことを目的として, このような設計となっている.

図 4 に init とその応答例を示す. この例では, 利用者が “/0/1/6” にあるノードを指定し, そこには Haskell[8] で書かれた “trackback root rpath cpath” というプログラムがあることが分かる.

```
<init/>

<result command="init">
 <item path="/0/1/6" children="T">
  <psdcode lang="haskell"
   >trackback root rpath cpath</psdcode>
 </item>
</result>
```

図 4: init とその応答例

2.3 ノード取得要求: get

get は要求中で XPath により指定したノードから, その属性以外の子ノードを全て除いたものを返す. 従ってある部分木全体が必要であるような場合, 一回の要求・応答ではその部分木のルートに当たるノードしか取得できないため, 再帰的に get を繰り返す必要がある.

get に対する応答にはそのノードの子ノードとしてどのようなものがあるかというヒントとして, 子ノードの種類のリストが含まれる. 応需 DOM ではノードの種類として要素 (E), テキスト (T), PI(P), コメント (C) 等が定義されており, 子ノードの出現順に各ノードに対応する文字の列として構成される. このヒントにより, 応需 DOM クライアントは DOM 木を効率的に巡回することが可能となる.

図 5 に get とその応答例を示す. この例では, “/0/1/4/3” の位置には “<body>” 要素があり, さらにその子孫としてテキスト, 要素, テキストのみっつがあることが分かる.

```
<get target="node"
     path="/0/1/4/3" type="BNDPath"/>

<result command="get">
 <item path="/0/1/4/3" children="TET">
  <body/></item>
</result>
```

図 5: get とその応答例

2.4 ノード更新要求: put

put は XPath により指定した場所に任意の部分木を挿入する. 挿入される部分木は要求中に XML エンコードされてサーバに送信される. この要求に対しサーバは成功したか失敗したかについて返答する.

図 6 に put とその応答例を示す. この例では, “/0/1/6” という位置にあるノードを “<psdresult>...</psdresult>” という木で置換している.

```
<put path="/0/1/6">
 <psdresult>
   <bref id="00077"/><bref id="04489"/>
 </psdresult>
</put>

<result command="put" path="/0/1/6"/>
```

図 6: put とその応答例

2.5 再帰的なコードの評価

PSD の枠組では, 必要に応じて外部評価系が呼び出され, 評価する必要のある部分が適時評価され, その結果に基づいて更新が行われる. そのため, 外部評価系は DOM の必要とされる部分に応需 DOM の get を用いてアクセスし, 計算を行う. そして最終的な結果が put で返され, 1 回の評価が完了する. このように通常は PSD の 1 回の評価・更新が応需 DOM の 1 セッションに対応する. しかし, PSD の評価中に再帰的に評価を行う必要が発生する場合もある. 応需 DOM サーバ側プロトコルでは再帰的な外部評価系の呼び出しに対応した機能は用意されていないため, 外部評価系側, またはそれらの間に配した制御プログラム等により再帰が起こったプロセスのセッション管理および通信の調停 (arbitration) を行わなければならない. この間, 特に更新されたノード以下のノードを他の応需 DOM セッションにおいて既に取得していた場合, そのノードに関して不整合が発生する. このような不整合を回避し, 一貫性を保証するのは外部評価系の責任である. 実際には外部評価系はノードの (部分木) の追加のみを行い, 更新 (差替) は行わないため, その実現は容易である.

3 Haskell 版応需 DOM クライアントの実装

本節では, 応需 DOM の実装例として Haskell 版クライアントの実装について述べる.

前節で述べたように, 応需 DOM で提供される基本的 API は要素取得 (get) およびノード更新 (put) のみである. 理論的にはこのセットで十分であるが, これだけではプログラミングには不便である. 例えば, オブジェクトモデルとして Java 等 DOM 抽象化を良く用いる言語では DOM に類したインタフェースが, また Haskell 等のようにデータ構造中心の表現が広く用いられる言語においては仮想的な木データとして扱えるようなインタフェースを提供することで, プログラミングの効率向上が可能となり, ひいてはより安全なプログラムをより素早く作成することができる. 応需 DOM の Haskell 版クライアントは Haskell のための応需 DOM プログラミング API を提供する. この API では DOM は仮想的な HaXML[5] のデータ構造としてユーザに提示される.

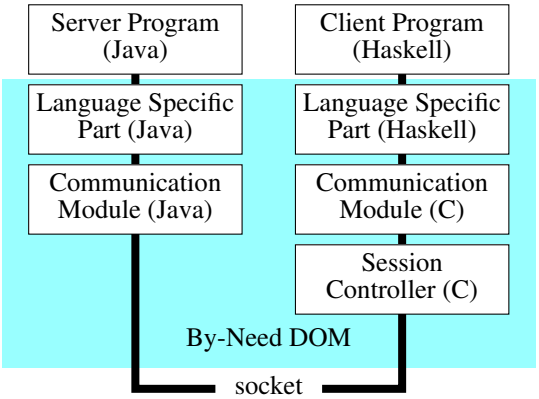

図 7: Haskell 版応需 DOM の構成図

前述のように, PSD の評価においてはその最中にコード (すなわち評価を行わなければ値が得られない部分) に到達することがある. その場合, 外部評価系は行っている評価を一旦中断し, その部分の評価を行った後に作業を再開しなければならない. 応需 DOM ではこのような再帰的処理のための機構は提供されないため, 外部評価系側でこのような再帰的処理に対応する必要がある. しかし, Haskell 版に限らず, クライアントライブラリ (図 7 Language Specific Part (Haskell)) でこの処理を行うとすれば, その部分が複雑になり, クライアントライブラリの移植性を損なうことになる. 一方, サーバ (図 7 Language Specific Part (Java)) 側にこの機能を組み込むとクライアントの見通しは良くなるが, サーバに複数クライアントとのセッション管理機能が必要となり, 応需 DOM サーバの移植性を損なうことになる. これは, PSD のように様々なエディタ等のプラグインとして提供することを前提としている場合, 大きな欠点である. そこで, 本実装では, サーバとクライアントの間に通信をリレーする通信サーバ (図 7 Session Controller (C)) を置き, この通信サーバにセッション管理機能を持たせるとともに, コードに到達したときの再帰的処理の管理を行わせることでこの問題を回避した. この部分は C 言語で書かれており独立して動作するもので, その移植は容易である.

Haskell にリンクされるクライアントライブラリから通信サーバへのアクセスはいずれも strict な操作であり, セッションの開始, 0 回以上の get, 0 回または 1 回の put, セッションの終了をこの順に行われなければならない. しかし, get については PSD の仕様より応需 DOM サーバ側が一貫した応答を行うと仮定して良く, 従って get への呼出は自由な順序で

行うことが可能である．Haskell 用ライブラリで, あるノードを指定すると再帰的にそのノード以下の部分木を get により取得する関数を定義する．この関数は Haskell の遅延評価機構によりユーザプログラムがアクセスした部分のみ評価され, よって必要最小限の get が呼び出される．

サーバと通信サーバおよびクライアントと通信サーバは応需 DOM に基づいたプロトコルを用いて通信する．サーバはコードについてそのノード以下を全て返すという特殊な処理を行うが, それ以外は, サーバもクライアントも get と put を基本単位とし, 必ず 1 往復で完結するというシンプルなプロトコルを保つことができている．なお, 本実装では通信サーバが pipe を通じて Haskell プログラムから put するデータ読み込むが, その読込が開始され以降その put が完結するまでは他の要求を受け付けないため, Haskell 側のコードは put するノードを書き出す以前に strict オペレータによりそのデータ生成で必要となる get を全て完了させている．この strict な動作は性能には影響しない．

クライアントのプログラムからは HaXML のデータ構造のみが見えており, Haskell での普通のプログラムとして作成することが可能である．また, 評価結果は HaXML のデータ構造で返すだけで良く, 実行時にランタイムルーチンがそれを自動的に put するため, put を明示的に呼びだすコード書く必要も無い．

4 関連研究

DOM を一度に全て読み込まず, 必要に応じて必要となった部分だけを処理するという試み [3] は広く行われている．なかでも応需 DOM に近いものに Lazy DOM [9] がある．これは, DOM メソッドが各ノードに初めてアクセスする際にそのノードを作成するというもので, DOM 内でのノードの展開を最適化することを目的としている．一方, 応需 DOM は DOM オブジェクトの必要部分にプロセス外からアクセスすることを目的としており, その点で大きく異なる．無論, 応需 DOM のサーバが保持している DOM オブジェクトが Lazy DOM のものであっても構わない．この場合, DOM が行うファイルアクセスと応需 DOM による DOM へのアクセスの双方が最適化される．

また, XML データベースなどでは応需 DOM と同様, ネットワーク経由でデータにアクセス可能になっているものも多い．そのうち多くは SOAP/HTTP により汎用かつ自由度の大きいデータ操作を提供している．それに対し応需 DOM は PSD の処理に特化した限られた操作しか行えないが, それによって簡素なプロトコルの実現が可能となった．

5 まとめ

PSD の処理を行う外部評価系がネットワーク経由で効率的に DOM オブジェクトにアクセスするための通信機構として応需 DOM を提案し, 実装を行った．Haskell 版クライアントでは DOM オブジェクトを仮想的な HaXML の木データとして提示することで, プログラマの負担を軽減し同時に Haskell の遅延評価機構によりデータアクセスが最適化される．また本稿では触れていないが, OCaml[6] 用クライアントも作成した．Haskell 版との大きな違いは通信サーバに相当する機能がクライアントライブラリに組み込まれており, それを必要としないことである．

現在, サーバ側プラグインは独自のエディタ用 1 種類しかなく, クライアント側ライブラリも Haskell 版と OCaml 版のみであるが, 今後, XMLSpy[10] 用プラグインおよび他の言語用クライアントライブラリを充実させて行く予定である．

参考文献

[1] V. Benzaken, G. Castagna, and A. Frisch, CDuce: An XML-Centric General-Purpose Language, Proc. the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP'03), 2003.

[2] Y. Hayashi, Z. Hu, M. Takeichi, N. Wake, M. Hara, and N. Oshima, Pruning DOM Trees for Structured Document Processing, 日本ソフトウェア科学会第 21 回大会, 2004.

[3] M. L. Noga, S. Schott, and W. Löwe, Lazy XML processing, Proc. the ACM Symposium on Document Engineering, 2002.

[4] M. Takeichi, Z. Hu, K. Kakehi, Y. Hayashi, S-C. Mu, and K. Nakano, TreeCalc: Towards Programmable Structured Documents, 日本ソフトウェア科学会第 20 回記念大会, 2003.

[5] M. Wallace and C. Runciman, Haskell and XML: Generic Combinators or Type-Based Translation?, Proc. the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP'99), 1999.

[6] http://caml.inria.fr/

[7] *Document Object Model (DOM).* W3C, http://www.w3.org/DOM/, 2000.

[8] http://www.haskell.org/

[9] ObjectWeb Consosium, Enhydra XMLC, http://xmlc.objectweb.org/

[10] Altova, Inc., XMLSpy. http://www.xmlspy.com/