JAIST Repository
https://dspace.jaist.ac.jp/

Title	XMLで表現されたCプログラムの静的解析ツールの設計と実現
Author(s)	川島, 勇人
Citation	
Issue Date	2002-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1532
Rights	
Description	Supervisor:権藤 克彦, 情報科学研究科, 修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

XMLで表現されたCプログラムの静的解析ツールの設計と実現

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

川島 勇人

2002年3月

修 士 論 文

XMLで表現されたCプログラムの静的解析ツールの設計と実現

指導教官　権藤克彦 助教授

審査委員主査　権藤克彦 助教授
審査委員　片山卓也 教授
審査委員　落水 浩一郎 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

010035川島 勇人

提出年月: 2002年2月

Copyright © 2002 by Hayato Kawashima

目 次

第 1 章　はじめに **1**
1.1 背景 1
1.2 目的 2
1.3 アプローチ 2
1.4 成果 3
1.5 本論文の構成 4

第 2 章　XML の概要 **5**
2.1 XML の勧告とその位置付け 5
2.2 XML の特徴 6
2.2.1 構造化データをテキスト形式で記述 7
2.2.2 マークアップ言語を定義 7
2.2.3 柔軟なデータ表現能力 8
2.2.4 安価で高度な関連技術 9
2.3 XML 文書の構成 11
2.4 XML 文書の処理 12
2.4.1 XML パーサ 13
2.4.2 DOM : Document Object Model 13
2.4.3 XSLT : XSL Transformations 14

第 3 章　CASE ツール開発の現状と XML 導入の利点 **16**
3.1 CASE ツール開発の現状 16
3.2 ソフトウェア開発環境における統合技術 16
3.3 データ統合技術の概要 17
3.4 既存のデータ統合技術 : CDIF と PCTE 22
3.4.1 CDIF : CASE Data Interchange Format 22
3.4.2 PCTE : Portable Common Tool Environments 24
3.5 XML による CASE ツール開発支援 27
3.5.1 XML の適用とその利点 27
3.5.2 上流工程を支援する XMI 28

第 4 章　XML を用いた CASE ツールプラットフォームの提案・実現　30
4.1 XML を用いた CASE ツールプラットフォームの提案 30
4.1.1 多種多様なソースプログラムの解析 31
4.1.2 プログラマの意図の理解 32
4.1.3 意味情報の管理・操作 33
4.1.4 上流・下流の統合 34
4.2 実現した CASE ツールプラットフォームの概要 35
4.2.1 ANSI C を対象にする理由 36
4.2.2 ACML : ANSI C Markup Language 38
4.2.3 XCI : Experimental C Interpreter 49

第 5 章　CASE ツール作成実験　54
5.1 スライシングツール 54
5.1.1 プログラムスライシングの概要 54
5.1.2 スライシングツールの実現 55
5.2 クロスリファレンサ 58
5.2.1 クロスリファレンサの概要 58
5.2.2 クロスリファレンサの実現 59

第 6 章　議論　61
6.1 開発コスト削減の要因 61
6.2 ACML 文書の解読 61
6.3 ACML の改良の余地 62
6.3.1 エレメントノードの文脈のための情報とリンク 62
6.3.2 前処理に関する情報 64
6.3.3 字句情報 (空白、コメント、インデント) の保存 64
6.3.4 DTD の決定性問題 64
6.4 ACML に対する操作・編集 65
6.4.1 DOM の利便性 65
6.4.2 XSLT の利便性 65
6.4.3 ライブラリの必要性 66

第 7 章　関連研究　67
7.1 Sapid 67
7.2 JavaML 69

第 8 章　おわりに　71
8.1 まとめ 71
8.2 結論 72

8.3 今後の課題 73

付 録 A ACML DTD **75**

参考文献 **79**

概 要

CASE ツールの開発は、各々が対象とするソフトウェアに応じた解析器を必要とするため、多くのコストがかかる。効率的な開発には、各々のツールで共通に用いられるデータの統合が有効である。そこで、我々は XML に注目した。なぜなら、XML は構造化文書のためのデータ記述フォーマットであるが、CASE ツールのデータ統合にも多くの利点を持つからである。例えば、XML の文書型定義 (DTD) を使うと、プログラムの複雑な構造や関係をコンパクトに表現できる。本研究では、XML を用いた CASE ツールプラットフォームを構築し、これを基に、プログラムスライシングツールとクロスリファレンサを作成した。この実装実験では、各々の実現は開発者 1 人で、わずか約 2 週間ですみ、開発コストの削減を確認した。

第1章　はじめに

1.1　背景

ソフトウェア開発の過程には、数多くのプロダクトが出現する。プロダクトとは、ソフトウェアを構成する種々の仕様図面や設計文書、プログラム、メモ、マニュアルなど、ソフトウェアを製品として仕上げるために必要となるあらゆる文書である。ソフトウェアの開発、保守の過程は、その多くがこれらプロダクトの参照と変更の過程である [42, 54]。

対象とするプロダクトが、複雑で規模が大きくなると、その参照、変更は、人手による作業だけでは限界があり困難である。そのため、コンピュータによる支援、つまり CASE[1] ツールが必要となる。しかし、一般に CASE ツール開発のコストは高い。その理由は、対象とするソフトウェアに応じて各々解析器を作成しなければならないからである。この解析器のデータは、各々の CASE ツール開発において、共通に扱うことができるのだが、往々にして、その CASE ツールの内部データを他の CASE ツールで使うことができない。よって、これらの CASE ツールは開放的でなく、ツール間の連携を取ることが難しい。

この問題を解決するために、各々のツールに共通に用いられるデータを統合しなければならない。開放的で効率的に CASE ツール間でのデータ共有やデータ変換を可能にする技術、つまり、データ統合技術を発見、開発することが大きな課題である。

データ統合のアイディアは、新しいものではない。現在までに構築されたデータ統合技術として、CDIF(CASE Data Interchange Format)[4] と PCTE(Portable Common Tool Environments)[3] がある。これらの技術は、データ統合に貢献してきているが、未だ CASE ツール開発に広く使われてはいない。

また、個々の CASE ツールのもつ機能とデータをまとめるために、CASE ツールプラットフォームが提案、構築されてきた。Sapid(Sophisticated Application Programming Interface for software Development)[45] は、その代表例である。CASE ツールプラットフォームとは、単なるツールの寄せ集めではなく、CASE ツールに共通なデータや機能を提供する基盤となるソフトウェアシステムである。これにより、ツール開発コストの削減とツール間の連携が容易になり、統一的で効率的なソフトウェア開発が期待できるが、現在も研究が進められており、CDIF や PCTE 同様、まだ実用的に使われるようになっていない。

これらを踏まえると、ソフトウェア開発におけるデータ統合技術は、本質的に困難であると確認することができる。それゆえ、様々な手法を試して、有用性を調べることが、データ統合技術の進歩において必要不可欠である。

[1] Computer Aided Software Engineering

1.2 目的

我々は、ソフトウェア開発における共通フォーマットとして、XML(Extensible Markup Language)[1] とその関連技術に注目した。

本研究では、XML を CASE ツールプラットフォームに適用し、その有効性を確認する。適切に設計した DTD(Document Type Definition) と、DOM(Document Object Model)[10] や XSLT(Extensible Stylesheet Language Transformations)[11] を始めとする XML 関連技術の適用が、CASE ツールの開発コストを大幅に削減すると期待できる。

我々の目標は、CASE ツールのための柔軟で使い易いデータ変換フォーマットの構築である。これは、CASE ツール開発を容易にする重要な要素である。特に、我々が注目するのは、XML による下流 CASE の統合である。なぜなら、ソフトウェア開発コストは、その大部分がテストや保守に費やされるからである。また、XML の下流 CASE への応用例は、まだほとんどない。

XML は構造化文書のためのデータ記述フォーマットであるが、CASE ツールのデータ統合にも、多くの利点を持つからである。その XML の主な利点を以下に示す。

- プログラム構文構造は、XML のエレメントの入れ子構造として自然に表現でき、プログラム要素間の関係は ID/IDREF リンクとして表現できる。
- 妥当な XML 文書として扱われないが、整形式な XML 文書として表現すれば、不完全なデータ (例えば、バグのあるコード) でさえも扱うことができる。
- XMI(UML 図の XML 表現形式) の標準化が進んでいる。下流 CASE ツールに XML を適用すれば、上流と下流の間のデータ統合が期待できる。

1.3 アプローチ

本研究では、XML を用いた CASE ツールプラットフォーム構築の第一歩として、ANSI C[2] プログラムのみをターゲットにした CASE ツールプラットフォームを構築した。XML が実用的に活用できるか確認するためには、我々は言語のサブセットではなく、フルセットをサポートすべきと考えている。実現した CASE ツールプラットフォームの構成要素は、次の 3 つである。

- **ACML : ANSI C Markup Language**
  ANSI C 用に設計した DTD で、データ統合のためのスキーマである。構文構造と静的意味情報を表現する。
- **XCI : Experimental C Interpreter**
  ANSI C プログラムを ACML 文書に変換するトランスレータ

[2] ISO/IEC9899-1990

- **XML 技術を基に開発した CASE ツール**
  アンパーサや、応用例として、プログラムスライシングツールとクロスリファレンサ

まず、ANSI C プログラムは、XCI によってタグ付けをし、ACML 文書に変換する。その ACML 文書は XML パーサを通じて DOM や XSLT で、プログラムスライスを抽出したり、ソースプログラムを表示したりするために処理する。

これを基に、CASE ツール作成実験として、プログラムスライシングツールとクロスリファレンサを作成した。これらの実験は、CASE ツールプラットフォームの有効性を計る例題として適切である。その主な理由は、これら 2 つの CASE ツールが、次に示す特徴を持つからである。

- 他の CASE ツールに比べ、非常に基本的、かつ必要性の高い静的解析ツールである。
- 解析において、プログラム要素間の関係を処理する必要がある。

1.4 成果

上述した実験において、各々の実現は開発者 1 人で、わずか約 2 週間ですんだ。これに対し、XCI の ANSI C の構文解析器と静的意味解析器の部分の実現には、開発者 1 人で、2 カ月を費やした。それゆえ、ACML と XCI を使ったことで、本来、必要であるはずの 2 カ月の開発期間を削減できたことになる。この種の小さな CASE ツールの作成においては、XML を用いる有用性を確認した。

また、我々は、これらの実験を通じて有益な経験をした。その主なものを 3 点、次に示す。

1. ACML には改良の余地があることを確認した。例えば、あるノードに対して、前方には幾つのノードが存在するか示す属性がない。これは、木構造内の走査、特に上方へ辿るときに非常に便利である。

2. DOM や XSLT の利便性は高かったが、CASE ツール開発において、共通に使うことができるライブラリを構築する必要性を確認した。例として、次の操作を挙げる。

   - 特定の識別子を持つステートメントをすべて抽出する。
   - 特定の識別子を含んだ最も深い部分のステートメントを抽出する。

3. DTD の利便性を再認識した。それは、ACML がデータ交換の簡潔な仕様として機能したので、プログラマ間のやり取りを少なくできたからである。実際に、XCI 開発者と CASE ツール開発者のやり取りは、ACML を定義した DTD のみで、ほとんどすんだ。

1.5 本論文の構成

本論文の構成は次の通り。

第 1 章 本研究の背景と目的として、CASE ツール開発におけるデータ統合の必要性があり、XML を CASE ツールプラットフォームに適用すること述べ、研究のアプローチとその成果の概略を示した。

第 2 章 XML は、構造化データをテキスト形式で記述できる汎用フォーマットで、特に Web 文書として注目を集め、広範囲で使われている。本研究においては、CASE データの中間フォーマットとして活用するが、まず、XML とその関連技術に関する概要をまとめる。

第 3 章 ソフトウェア開発環境における 4 つの統合技術 (データ統合、制御統合、ユーザインタフェース統合、プロセス統合) を示し、その中で、特に注目するデータ統合技術について述べ、応用例 (CDIF、PCTE) を示す。これを踏まえ、XML を CASE ツール開発に適用する利点を列挙する。

第 4 章 目標とする XML を用いたソフトウェア開発環境と、その第一歩として、ANSI C にターゲットを定めて実現した CASE ツールプラットフォームについて述べる。特に、データ統合のためのスキーマである ACML に関する詳細 (DTD 設計、ANSI C の表現方法、開発要因など) を示す。

第 5 章 実現した CASE ツールプラットフォームの有効性を示すために行なった 2 つの CASE ツール作成実験、プログラミングスライシングツールとクロスリファレンサの実現について述べ、その結果を示す。

第 6 章 CASE ツール作成実験の結果から、開発期間を約 2 カ月削減できた要因を示し、この実験を通じて得た有益な経験、例えば、ACML の改良点や DTD の規格上の問題、XML 技術の利便性などについて議論する。

第 7 章 関連研究として、細粒度リポジトリに基づいた CASE ツールプラットフォーム Sapid と、XML を使って Java 構文規則から独立して Java プログラムをモデル化しようと試みている JavaML について述べる。また、本研究との相違を述べる。

第 8 章 本論文についてのまとめを行ない、結論として、XML が CASE ツール開発コストを削減したことを述べ、XML がデータ統合技術にとって有用であることを示す。最後に将来への課題をまとめる。

第2章　XMLの概要

本研究では、XML を CASE ツール開発に適用する。そこで、本章では、XML とその関連技術についての概要を述べる。

2.1　XMLの勧告とその位置付け

XML(Extensible Markup Language) は、World Wide Web コンソーシアム (W3C) によって 1998 年 2 月に勧告された新しい文書フォーマットである。XML は、次の設計目標 [1] に基づいている。

1. XML は、インターネット上でそのまま使用できる。
2. XML は、広範囲のアプリケーションを支援する。
3. XML は、SGML[1] と互換性をもつ。
4. XML 文書を処理するプログラムは容易に書ける。
5. XML では、オプションの機能はできるだけ少なくし、理想的には 1 つも存在しない。
6. XML 文書は、人間にとって読みやすく、十分に理解しやすい。
7. XML の設計は、速やかに行う。
8. XML の設計は、厳密で、しかも簡潔なものとする。
9. XML 文書は、容易に作成できる。
10. XML では、マーク付けの数を減らすことは重要ではない。

XML は SGML の後継言語として位置付けられる。SGML は、文書を記述するための言語で、ISO によって 1986 年に国際規格として制定された。SGML は、XML よりも以前から存在しており、自由にタグを決めることで、マークアップ言語を作り出すための枠組であるという点でも、XML と同様である。しかし、SGML は複雑で大規模なため、実装す

[1] Standard Generalized Markup Language

ることも使いこなすことも困難である。実際、SGML は電子的な文書管理という限られた用途にしか普及していない。

一方、広く普及した HTML[2][64] は、SGML に基づいて<h1>や<p>などのタグを 定義して作られたマークアップ言語で ある。HTML が普及した大きな理由は、誰でも使えるように、見出しや箇条書、段落などを表現する簡単なタグを整備したことにある。しかし、その反面、HTML は拡張性を持たない言語となった。

XML もまた、SGML を基にして設計された言語である。独自タグを導入してマークアップ言語を作り出すという利点を継承しつつ、SGML の複雑さを大幅に簡略化した。また、XML は HTML の長所 (例えば、URL による WWW の利用) を取り入れている。つまり、XML はSGML と HTML の成果を結集したものである。以下に SGML と HTML、各々から継承した主な長所を挙げる。

- SGML から継承した長所
  - タグの種類を自由に定義することが可能である。
  - 構造化文書の作成に対応する。
  - 強力な汎用スタイルシート (例えば、DSSSL[3] など) の使用が可能である。
  - 厳密な文法チェックが可能であり、電子化が容易である。
- HTML から継承した長所
  - 読み易い簡単なフォーマットである。
  - ハイパーリンクを装備する。
  - Web 技術 (例えば、URL や MIME など) に対応する。
  - スクリプト処理やプログラミングによる操作が容易である。

2.2 XML の特徴

XML は、汎用的なデータ記述言語として標準的なデータ記述フォーマットを提供するデータ表現技術である。XML が優れている点は、主に次の 4 つである。

- 構造化データをテキスト形式で記述できる。
- マークアップ言語を定義する機能がある。
- 柔軟なデータ表現能力がある。
- 安価で高度な関連技術が多い。

[2]HyperText Markup Language
[3]Document Style Semantics and Specification Language

2.2.1 構造化データをテキスト形式で記述

XML は、データの意味を表すタグ名と、入れ子による論理構造化によって、データの構造とその意味を保持したまま、データ交換できる。XML データはテキストなので、HTML と同様、インターネットの通信プロトコルで送信するのに向いている。また、アプリケーション開発において、あるバイナリーフォーマットで記述している場合、例えば、データが「メッセージの先頭からのオフセット 12 バイト目から 4 バイトにネットワークバイトオーダーで記述されている 4 バイト整数」という形で記述されているならば、このメッセージを理解するためには、16 進ダンプを眺めなければならない。これに対し、XML はテキストベースであり、そのままの形で端末に表示して眺めることができる。一般のテキストエディタや、UNIX のテキスト処理ツール (例えば、grep など) を用いて処理することもできる。これにより、XML 文書を読んだり、作成したり、編集したりすることが特別なツールなしで可能なのである。このように、XML は構造化データをテキスト形式のマークアップ言語で記述するのである。

2.2.2 マークアップ言語を定義

HTML のタグは名前が固定的であるが、XML は用途に合わせてタグの名前を定義して使用できる。XML では、データで使用するタグ名、タグの階層構造、そしてタグで囲まれる内容データのデータ型などを定義できる。この XML のデータ設計のことを XML のスキーマ (schema) という。XML では、スキーマを設計して定義し、それに基づいて XML のデータを作成する。XML のスキーマは、XML データで使用するタグ・セットを定義する。これにより、データがスキーマを持ち、それに基づいて厳密なデータ交換を行なうことができる。環境の異なるアプリケーションがデータを交換するのに適している。同じ意味を保持したプラットフォームに依存しない XML のデータを交換できることから、XML はデータの相互運用性 (interoperability) を確保する。

XML がスキーマでタグ・セットを定義することにより、事実上、XML に準拠した特定目的のマークアップ言語を定義できることを意味する。このように、XML はメタ言語としての機能を備えることから、用途の面で柔軟性を持ち、データ記述の汎用性を確保している。よって、XML には言語として次の 2 つの側面がある。

- 構造化データを記述するマークアップ言語
- マークアップ言語を定義するメタ言語

XML は、メタ言語とマークアップ言語、この両方の機能を持つ言語である。

2.2.3　柔軟なデータ表現能力

XML は、本質的に木構造のデータの表現が可能である。木構造は、多くのアプリケーションに対して十分に強力であり、また概念と処理が簡単なので、XML の基本データ構造となっている。

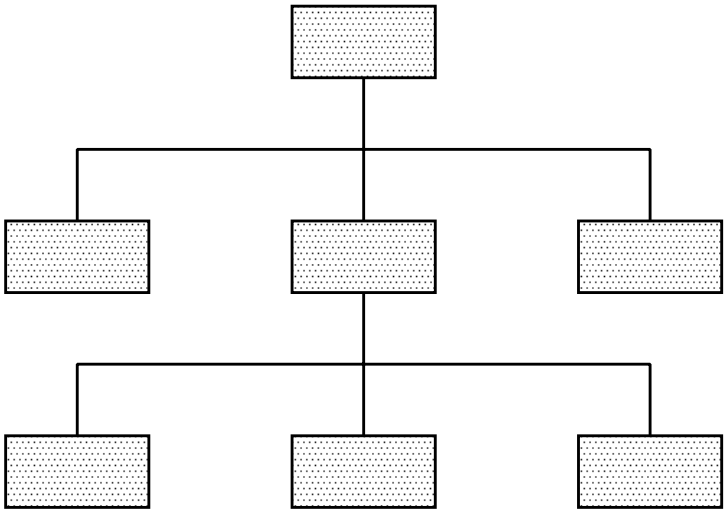

図 2.1: 木構造

これ以外のデータ表現としては、例えば、表構造やグラフ構造が表現できる。表構造は、関係データベースの基本的な表現形式であり、XML で表を表すには、行を 1 つの部分ツリーで表すか、あるいは、属性を用いて 1 つのエレメントを 1 つの行として表す必要がある。

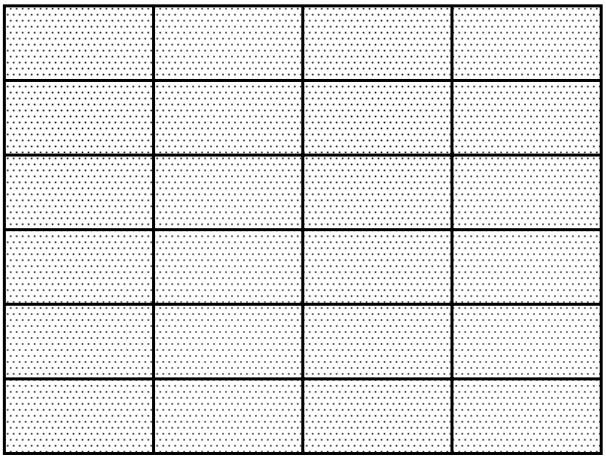

図 2.2: 表構造

グラフ構造には、幾つかレベルがあるが、例えば、サイクルを含むグラフの表現は、リッチな表現である。XML に基づいてグラフを表現するには、ID/IDREF 属性を用いるか、あるいは RDF[4][14, 15] を用いることが必要である。

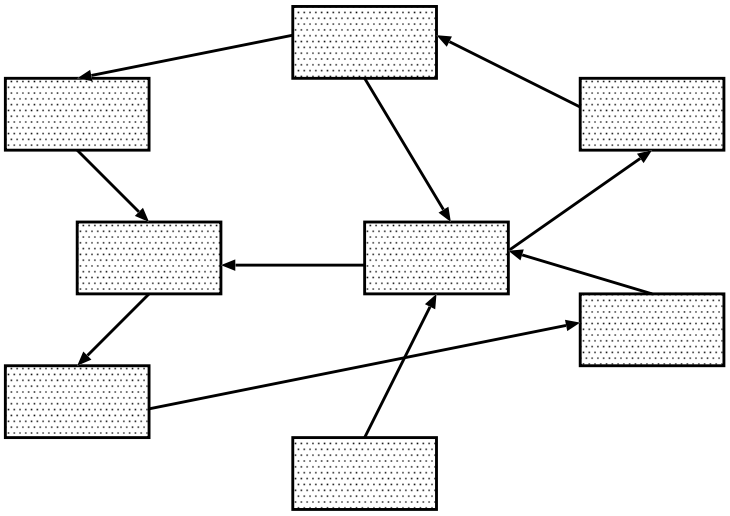

図 2.3: グラフ構造

XML は、設計次第で多種多様なデータ構造を柔軟に表現できる。直観的には、XML 文書の論理構造から、木構造が表現し易い。厳密な文法チェックにより、保証された入れ子になったエレメントが自然に木構造になる。これに対して、グラフ構造は、エレメント内の属性を適切に用いることで実現する。この場合、属性に対しても厳密な文法チェックが施されているため、リンク切れがないことが保証されている。テーブルに関しては、入れ子になったエレメントの表現を用いることも、属性を用いることも可能である。また、これら 3 つのデータ構造は、独立して表現するだけでなく、同時に表現することも可能である。これは XML の持つ大きな利点であり、我々の目的にとっても、非常に重要な特徴である。

2.2.4 安価で高度な関連技術

XML の広範囲に渡る実用化が進んだ背景には、XML の力を引き出す関連技術の開発や整備が充実していたことにある。XML の関連技術は、W3C を中心に規格の開発が行なわれ、すぐにベンダー各社や、個人の有志が実現する。それらの多くは、安価で高度なものである。ここでは、1. 表示、2. 変換、3. プログラミング、4. スキーマ定義、5. リンク、に分けて概要を示す。

[4]Resource Description Framework

1. **XML の表示**
   XML のデータを表示する方法には、基本的に次の方法がある。

   (a) CSS[5][18] でスタイルを定義して表示する。

   (b) XSLT[11] で HTML に変換して表示する。

   (c) XSLT で XSL[19] のフォーマッティングオブジェクトに変換して表示、印刷する。

2. **XML の変換**
   XML データを別の形式の XML データに変換するには、XSLT を使用できる。XSLT は、XML のデータを他の形式の XML データに変換する規則を定義する言語である。XSLT では、XML のデータの一部を特定するために、XPath[12] という規格を使用している。この XPath は、XML データの一部を特定するための記述方法を与えるものである。

3. **XML のプログラミング**
   XML のデータを操作する方法には、次の 2 つの方法がある。

   (a) XML のデータをすべて読み込んで DOM[10] ツリーというオブジェクトを作成し、DOM ツリーへの操作 API[6]を使用する。

   (b) XML のデータを読み込みながら、検出したタグやデータをイベントとして返して、その都度、処理できるようにする SAX[7][21] インタフェースを使用する。

4. **XML スキーマ定義**
   XML のスキーマを記述する方法には、XML1.0 が規定している DTD を使用する方法と、データ型を強化すると共にスキーマ自体も XML データとして記述する XML Schema[16] を使用する方法がある。これら以外にも、この分野では、多くの規格案 (例えば、RELAX[17] など) が登場している。また、XML データ記述でスキーマをさらに生かすための名前空間 (XML Namespace)[13] 機能がある。

5. **XML のリンク**
   XML データにリンクを設定する規格に XLink[23] がある。これは、XML でのハイパーリンク機能を規定し、使用するポインタが XPointer(勧告候補)[22] である。この XPointer は、XLink のリンクで使用するポインタの記述を規定し、XPath を使用する。

XML には、これら以外にも多くの関連規格 (例えば、XML データの照会言語である XML Query[24] など) がある。

[5]Cascading Style Sheet
[6]Application Programming Interface
[7]The Simple API for XML

XML は、単にデータの書き方を定めたもので、パース以降のデータ処理に関しては、何も規定してない。そのため、上で示した XML 関連規格を始め、多くの周辺技術によって XML は支えられている。

2.3 XML 文書の構成

XML 文書は、大きく分けて次の 3 つの部分から成り立っている。

- **XML 宣言 (XML declaration)**
  XML のバージョン宣言、文字コードの宣言などを行なう。これは、省略可能な部分である。

- **DTD(Document Type Definition : 文書型定義)**
  DTD は、XML 文書で使用されるタグの種類やその属性、階層構造、出現順序などを定義したものである。DTD は、拡張 BNF 記法を用いて記述することになっている。DTD の基本要素は次の通りである。

  - **エレメント型宣言 : element type declaration**
    XML において、エレメントは 1 つのデータを表すものである。ここでは、XML 文書中で使用できるすべてのエレメントの名前を定義し、出現順序や階層構造、繰り返し回数を指定する。
  - **属性リスト宣言 : attribute-list declaration**
    属性は、エレメントに関する補足的な情報を提供する。ここでは、XML 文書中で使用できる各エレメントについて、属性名、属性型、デフォルト値を定義する。属性型の例として、ID/IDREF 型の属性がある。これは、エレメントを一意に識別するための ID 型の属性値に対して、IDREF 型の属性値を使うことで、ID 番号による参照を可能にする。
  - **エンティティ宣言 : entity declaration**
    エンティティは、例えば、ファイルや置換文字列のように、何らかの形で XML のデータの一部を格納するものである。ここでは、XML 文書や DTD の中で使用するすべてのエンティティの名前と内容 (例えば、置き換えたい文字群や外部ファイル) を定義する。
  - **記法宣言 : notation declaration**
    外部ファイルとして参照するエンティティが XML 以外の記法 (例えば、Tex や EPS、GIF など) を使っている場合、その記法を識別するための名前を指定する。

- **XML インスタンス**
  実際のタグ付きの文書が記述される部分である。

これを基に、XML 文書は、処理上の観点から 2 つのレベルに分類することができる。1 つが整形式の XML 文書 (Well-Formed XML Document) と、もう 1 つは妥当な XML 文書 (Valid XML Document) である。

- **整形式の XML 文書**
  開始タグと終了タグの対応が取れていて、親子関係にあるエレメントのタグは、正しく入れ子関係になっているなど、XML で規定したタグ付け規則に従って XML インスタンスが記述されている XML 文書のことである。そのため、整形式の XML 文書は DTD を必須としない。

- **妥当な XML 文書**
  DTD の中の要素宣言、属性リスト宣言によって定義されたエレメントの階層関係、属性の型などに従ってタグ付けが行なわれている XML 文書のことである。当然、整形式の XML 文書としてのタグ付け規則に従っていることが前提条件である。

整形式の XML 文書が許されることは、XML の大きな特徴の 1 つである。SGML においては、基本的に、DTD を明示することが原則であった。しかし、必ず明確な DTD を示さねば処理できないというのは煩雑であり、また、いちいち DTD に適合するかどうかチェックするのは処理が重くなり、ソフトウェアも複雑になった。そこで、これらの問題を解決するために、XML では、DTD を必須としない整形式の XML 文書を導入したのである。

また、この整形式の XML 文書が許されたことにより、1 つの XML 文書に種々の DTD のエレメント型や属性名を記述しても実用上問題ないものになった。しかし、複数のタグセットに属するエレメント型や属性名を混在させたとき、タグ名や属性名が重複してしまう場合がある。この場合、タグや属性の名前が同じなので、どちらのアプリケーションが処理を担当しなければならないのか判別ができなくなる。この問題を解決するために、XML Namespace が導入された。これは、エレメント型や属性名がどの DTD に従うものなのかを識別する仕組みである。これにより、多種多様なアプリケーションで使用するエレメントや属性を混在させることが可能になったのである。

2.4 XML 文書の処理

XML 文書を処理する技法のうち最も普及したものとしては DOM、SAX、および XSLT がある。これらを含め XML 文書の処理は、幾つかのステップを踏んで行なう。例えば、パースやスタイルシートの適用、Web ブラウザを使っての表示など、これらのステップがすべて組合わさって、全体として XML 文書の処理が成立する。ここでは、次に示す XML 文書の処理で多く用いられる XML 関連技術について述べる。

1. XML 文書の構造を調べ、結果を XML アプリケーションに渡す役目をする XML パーサ。

2. XML パーサから受け取った情報を操作、編集する API である DOM。

3. XML パーサから受け取った情報を別の形式に変換する規則定義する XSLT。

2.4.1 XML パーサ

XML パーサとは、XML 文書を読み込み、その構造と内容へのアクセスを提供するために使用するソフトウェアである。XML 文書を読み込む際には、整形式な XML 文書としてのチェックを行ない、必要であれば (DTD があれば) 妥当な XML 文書としてのチェックも行なう。XML 文書はテキストファイルとして存在しているが、XML には独自のタグや属性を定義することが許されているため、XML を処理するアプリケーションを作成することは、容易ではない。しかし、XML 文書の書式は決まっているため、XML 文書を読み込んで、それをアプリケーションに渡すまでの処理は、共通のものとなる。この共通する作業を行なうのが、XML パーサである。

XML 文書を扱うアプリケーションを作成する場合には、XML 文書を直接扱わず、XML パーサを経由させることが一般的である。アプリケーションから XML パーサを呼び出す場合には、主に標準化された API(例えば、DOM や SAX など) を使用する。

XML パーサは、各ベンダーや個人の有志により開発され、その多くを無料で使うことができる。主な XML パーサを以下に示す。

- XML for Java Parser(IBM)[25]
- JAXP(Sun)[26]
- Xerces Java Parser(Apache)[27]
- XP(James Clark)[28]
- Lark(Tim Bray)[29]

XML パーサは、上で示した他にも多く存在し、各々特徴を持っている。そのため、開発するアプリケーションによって、最適な XML パーサを選択することができる。

2.4.2 DOM : Document Object Model

XML 文書をアプリケーションで利用するための API の 1 つで、W3C が公式に公開した唯一の API である。DOM は、XML 文書を「DOM ツリー」と呼ばれる木構造として扱う。そのため、XML パーサが XML 文書全体を読み込んだ後でなければ、文書内のデータにアクセスすることができない。DOM の特徴の 1 つは DOM オブジェクトに対する更新を行えることである。DOM オブジェクトを XML 文書として取り出せるので、DOM オブジェクトを経由して XML 文書の更新ができることになる。DOM のもう 1 つの特徴は、XML 文書の内容が DOM ツリーとしてメモリ上に展開されるので、XML 文書内のデータの順番に関係なくアクセスできる。これにより、DOM ツリーを縦横無尽に移動すること

で、複雑な構造の XML 文書でも容易にアクセスすることができる。このように、DOM は、XML 文書の参照や変更を可能にするものである。

DOM は公開された標準であるため、これに準拠した XML パーサが複数公開されている (2.4.1 節)。XML 文書から DOM ツリーへの変換と DOM ツリーから XML 文書への変換に関しては、XML と DOM の仕様書 [1, 10] では、明文化されていないため、パーサ依存のコードを記述しなければならないが、それ以外の部分は、DOM 準拠のコードを記述しておけば、異なるパーサの間でコードを共有できる。このため、XML パーサの選択の幅が広がり、開発の途中で XML パーサを入れ替えることも容易にできる。

DOM は、XML のすべてをオブジェクト指向プログラミングの枠組みに合致させる。まず、DOM では、形式的に IDL[8](インタフェース定義言語) を使用して、オブジェクトへのアクセスと操作のためのインタフェースを定義している。これは、どのプログラム言語よりも高度な抽象化を行ったことになる。次に、具体的な言語の実装の定義は言語バインディング (Language Binding) によって指定する。ここで、IDL が指定したデータ型や属性、メソッドの定義をプログラミング言語にあった方法で定義しなおす。よって、DOM は言語に依存せず、その”文書オブジェクト・モデル (Document Object Model)” は、多数の汎用プログラム言語 (例えば、Java や C++、Perl など) のライブラリとして提供される。DOM はすべてのメソッドと引数を指定するため、言語に近いものである。ライブラリの基礎となる汎用言語は単なる接着剤の役割を担うに過ぎないのである。そのため、どんな開発言語を使用しても DOM の API は同じで、DOM ツリーの構造も同じなので、異なった開発言語間で共通のプログラミングモデルを使用できるという利点がある。

2.4.3 XSLT : XSL Transformations

XML のスタイルシート言語である XSL には、本来、変換言語としての機能と、実際にテキストをフォーマットする言語の 2 種類の言語が含まれていた。この中で、変換言語の部分は非常に有用で、XSL 以外の目的でも役に立つことから、分離独立し XSLT として、現在、W3C の勧告となっている。

XSLT は、任意の XML 文書を読み込んで、それを加工して出力するスタイルシート変換用の言語として使用することができる。出力は XML 文書とは限らず、他のフォーマット (例えば、プレーンテキストや HTML) を出力することができる。そのため、XML 文書を HTML に変換するために使用されることも多い。

XSLT スタイルシートの構造は、XML 文書の構文を使用して表される。このため、XML の語彙処理の機能 (例えば、Unicode 文字のエンコードとエスケープ、外部エンティティの使用など) が、すべて使用できる。

基本的な処理パラダイムはパターン・マッチングである。XSLT スタイルシートは、各々が「この条件が入力で満たされたら、次の出力を生成する。」という形式を取る一連のテンプレート規則で構成されている。規則の順序は重要でなく、複数の規則が同じ入力に一

[8]Interface Definition Language

致する場合は、競合解決アルゴリズムが適用される。入力 XML 文書は木構造として扱われ、各テンプレート規則は木構造内のノードに適用される。次に処理するノードはテンプレート規則自体が決定できるため、入力は必ずしも元の文書における順序ではスキャンされないのである。

XSLT では、入力する木構造内のノードを参照するのに XPath というサブ言語を使用する。XPath は、本質的に、XML の階層データモデルに適した照会言語である。これには、木構造を任意の方向にナビゲートしてノードを選択したり、ノードの値や位置に基づいて述部を適用したりする機能がある。また、基本的なストリング操作、数値計算、およびブール代数のための機能も組み込まれている。例えば、XPath 表現 `../@title` では、現在のノードの親であるエレメントの `title` 属性が選択される。XPath 表現を使用すれば、処理用の入力ノードの選択、条件付き処理における条件のテスト、および出力する木構造への挿入用の値の計算を行うことができる。テンプレート規則では、特定のテンプレート規則が適用されるノードを定義するために、単純化された形式の XPath 表現であるパターンも使用できる。

XSLT は、関数型言語 (例えば、Lisp や Haskell、Scheme など) の概念に基づいている。スタイルシートは、本質的に純粋な関数であるテンプレートで構成されている。各テンプレートは、出力する木構造の断片を、入力する木構造の断片の関数として定義しており、いかなる副作用もない。よって、言語に再帰機能が備わり、処理内容は、かなり明確に関数型として表現できる。しかし、XSLT は関数型プログラミングの着想に基づいているが、関数を第 1 クラスのデータ型として扱う機能が欠けているため、まだ関数型プログラミング言語ではない。このため、XSLT は万能ではなく (例えば、比較機能が少ない点)、あくまで、スタイルシート変換用の言語であって、すべてのニーズに対応することはできない。例えば、子ノードの内、素数番目のものだけを抽出することはできない。

第3章　CASEツール開発の現状とXML導入の利点

3.1　CASEツール開発の現状

CASE ツールは、対象ソフトウェアに応じて各々解析器を持たなければならず、その解析器の作成を含め、開発には多くのコストがかかる。しかし、この解析器の作成は CASE ツール開発において必要なものであるが、解析器の作成自体は、目的とする CASE ツールの機能を実現する本質的な作業でないことが多い [47]。

また往々にして、これらのツールは開放的でなく、ツール間の連携を取ることが、念頭に置かれていない。それは、たいていの場合、その CASE ツールの内部データを他の CASE ツールで使えないことによるものである。つまり、非開放的であるか、開放的であっても、そのデータが使いにくいため、コスト高となる。

例えば、GCC[8] の内部で処理されるシンボルや構文、型情報などは、他のツールで使用することが容易にはできない。また、仮にできたとしても、内部表現に依存するので保守性は悪くなる。

よって、各々のツールに共通に用いられるデータを統合しなければならない。このためには、開放的で効率的に CASE ツール間でのデータ共有やデータ変換を可能にする技術を発見、開発することが大きな課題である。

3.2　ソフトウェア開発環境における統合技術

CASE ツールのための共通フォーマットのアイディアは、新しいものではない。ここでは、一般的な統合技術について述べる。ソフトウェア開発環境に開放性を与えるのは、CASE ツールの持つ機能と情報を取りまとめ、一定の共通性を設定する統合技術である。以下に開発環境における 4 種類の統合機能を挙げる [54]。理想的な開発環境は、これら４つの機能を同時に満たすものである。

- **データ統合**
  仕様記述やスクリプト、プログラムなど、プロダクトデータの要素や構成、及び、その扱い方を共通的に規定する。

- **制御統合**
  ツールの起動・終了やデータアクセスなど、共通的なツールの動作と通信の方法を規定し、ツール間の連携・協調を図る。

- **ユーザインタフェース統合**
  ツールのユーザインタフェースを共通化し、開発環境の使用性を高める。

- **プロセス統合**
  ツールによって支援される開発プロセスを共通的に想定し、個々の開発作業・工程を繋ぐ。

CASE ツールの統合化には、これら 4 つの統合技術が必要である。データ統合により、あるツールで作成した情報が他のツールでも共通に使用することができる。制御統合により、ツールの起動や終了、同期、協調動作などのツール間での通信が円滑に行なえるようになる。ユーザインタフェース統合により、ユーザインタフェースをツール間で共通化して使い勝手を良くすることができる。プロセス統合により、あるプロセスの実行結果に基づいて次のプロセスを選択するなど、定義したプロセスに従ったツールの利用ができる。

3.3 データ統合技術の概要

上述した 4 つの統合技術のうち、我々は特にデータ統合に注目する。なぜなら、CASE ツール開発のコスト削減には、データ統合が最も重要だと、我々は考えているからである。その理由は大きく 2 つある。

1. データ統合はプロダクトの最も根幹を統合するので、開発コストに密接に関係する。例えば、改行コードが 1 つに統合されれば、異なる改行コードの処理コードが不要になり、開発保守が容易になる。

2. それにもかかわらず、データ統合は他の 3 つの統合に比べて遅れている。既存の開発環境の多くは、制御統合やユーザインタフェース統合しか提供してない。

以下では、データ統合を実現するための 2 つのアプローチと、データ統合と他の 3 つの統合技術の関係を示し、データ統合の基盤となるリポジトリとその中に格納する情報について述べる。

データ統合へのアプローチとその位置付け

データ統合においては、次の 2 次元の多様性を統合する必要がある。

- **ツールの多様性**
  同種のプロダクトを扱うものでも、ツールごとにプロダクトデータの表現形式が異なる。

- **プロダクトの多様性**
  ソフトウェア開発の各工程では、多種多様なプロダクトが現れ、その要素や構成が異なる。

そして、このような多様性に対処するアプローチにも 2 種類ある。

- **共通フォーマット**
  構造や意味を規定する共通のデータ記述フォーマットを設定し、多様性を吸収する。つまり、ツール間で情報をファイルでやり取りする方法である。一方のツールが、その処理結果をあるフォーマットのファイルに出力し、他方のツールがこのファイルを入力する。この方式は特定の 2 つのツール間での約束だけで実行できるため、現実的であり実例 (例えば、CDIF[4](3.4.1 節)) が多い。

- **スキーマ定義**
  多様なものを多様なりに、その各々のスキーマが定義できる方法を提供する。つまり、ツール間で共通なリポジトリを介して情報交換を行なう方法である。この方式では、スキーマ定義に準拠したツール間であれば異なるベンダーの製品であっても、リポジトリに蓄積された情報を共有できる。標準化案としては、PCTE[3](3.4.2 節) がある。

ツールの多様性については、同種のプロダクトを扱うという共通点があるので、そこに含まれる意味的な要素や、その構成法がある程度のバリエーションはあるとしても、およそ範囲が定まっており、共通フォーマットで対応することもできる。しかし、プロダクトの多様性に対応しようとすると、このような共通フォーマットを網羅的に用意しなければならない。

スキーマ定義は、この多様性を網羅する問題に立ち入らず、幅広い対応が可能であることが特長であるが、プロダクトの扱いが構文的に留まってしまうのが普通である。言い換えれば、共通フォーマットが中身の問題であるのに対して、スキーマ定義は器としてのサービスが中心である。ここで、中身とは、データやメッセージの意味的構成を形作る機能であり、器とは、それを蓄積したり伝送したりする機能である。中身サービスが主として技術的支援であるのに対し、器サービスには、管理機能を支える側面がある。

また、データ統合は他の統合機能のベースとなる。制御統合との関連で見ると、制御統合を必要とするイベントの発生源を辿れば、プロダクトや、その要素に由来するし、制御の結果もプロダクトに何らかの影響を与えて、初めて意味を持つ。ユーザインタフェースやプロセスについても同じことが言える。

データ統合の基盤となるソフトウェアリポジトリ

データ統合の基盤となるのが、ソフトウェアリポジトリである。これは、簡単に言えば、CASE のためのデータベースであり、上で述べた器に相当する。ソフトウェア開発環境において、データや情報の扱い方は、段階を踏んで発展してきた。この発展段階を整理すると、順に示す次の 3 項目になる [54]。

1. **プロダクトライブラリ**
   多種多様なプロダクトを電子化したファイルとして格納、管理する。例えば、バージョン管理システムは、この形態である。プロダクトはファイルを単位として管理する。検索や解析を (ファイルより下位の単位である) 行単位で行なうことはできるが、意味を考慮した作業は人任せになる。

2. **データ辞書**
   管理の単位が細かく、かつソフトウェアの意味を反映したものとなる。種々の仕様図のノードやアーク、プログラムの変数や関数などの要素の名前や属性を管理する。辞書と呼ばれるように、これは、2 次データであって、1 次データであるプロダクトそのものは別個に存在する。プロダクト自体は、プロダクトライブラリと同様にストリームデータのファイルである。

3. **リポジトリ**
   プロダクトそのものを構造化して格納する。したがって、細粒度の意味的要求が識別でき、その属性が管理できるだけでなく、要素間の関係が扱いやすい。プロダクトのトラバースによる解析を行なうには、このリポジトリは必須となる。データ辞書が提供した情報は、1 次データであるこのプロダクトのインスタンスにすべて含まれる。リポジトリにおける 2 次データは、さらに上位 (メタ) の情報、すなわち 1 次データの要素の型を与えるスキーマであり、リポジトリのシステムに内包される。これにより、多種多様なプロダクトを包括的に扱うことができる。

ソフトウェアの開発、保守には、ファイルよりも細かい単位の構成管理や、修正、変更を伴い、プロダクトを構造化して格納できるリポジトリは、有用である。このリポジトリを実現するためには、一般にデータベース技術が必要になる。そこで、CASE 環境において期待できるデータベース技術として、例えば、次に示す 2 点がある。

1. ソフトウェアは非常に複雑な構造を持つので、複合オブジェクトを扱えるオブジェクト指向データベースが必須である。

2. ソフトウェアの構成要素間の関係は非常に複雑なので、リッチな意味モデルが必要である。

リポジトリは、ソフトウェアの意味的構造を扱えようにするため、その構成要素間の関係が複雑になり、その複雑さに適応できる上のような技術が必須になる。しかし、これらの技術には、複雑な要素関係を扱えるという利点があるのに対して、各々次に示す問題点がある。

1. 複雑な構造を複雑な複合オブジェクトに組み上げて、ナビゲーションがうまくいくか、メソッド連鎖の設計や操作が困難である。

2. 複雑な構造の上に、さらに多種多様なリンクを張りめぐらして、関係解析がうまくいくか、関係記述の作成や保守に多大な工数がかかる。

このように、リポジトリはソフトウェア開発、保守には有用であるが、あまりに複雑な要素関係を扱わなければならず、実現するには困難な点が幾つかある。PCTE[3] は、リポジトリの機能を実現した実例であるが、問題点があり、広く普及するに至ってない (3.4.2 節)。そのため、今後もソフトウェアリポジトリには、多くの技術を適用する試みが必要である。

ソフトウェアの意味情報

ソフトウェアの意味情報は、リポジトリを器と例えると、そこに盛る、上述したデータ統合の中身に相当する。リポジトリで仕切り方が決まり、それによって各々の仕切りの中に入れるべきものもある程度規定されるが、ソフトウェアの意味を的確に表すためには、まだ粗すぎる。構文論的言えば、数少ない非終端記号が多くの終端記号を一挙に生成するような状態である。中間的な非終端記号を適度な数に増やすことが必要である。

適度な数の非終端記号とは、ソフトウェアの意味的要素を適度に包括したものであるが、これでは、あまりに曖昧である。ここでは、この曖昧さを軽減するために、ソフトウェアに対してトップダウンにアプローチする参照モデルを示す。

その参照モデルとは、IEEE(P1175)[40] の CASE ツール相互参照モデルのことである。この参照モデルでは、ツール相互接続のための要件や環境を組織やプラットフォームとの関連も含めて整理した上で、ツール間の情報交換について解き進めている。情報交換のメカニズムやプロセスについてまとめた後、伝達される情報そのものが議論される。

- **制御情報**
  ツール間の通信制御プロトコル (通信の開始、終了、送信、受信、再送など) を定める情報。ツールプラットフォームにおいて定められる。

- **管理情報**
  主題情報 (次項) を管理するための情報。

  – 品質情報:データ構造の複雑さなど。

- 構成・版管理情報:開発開始終了日付、開発部署、開発者など。
- 計量指標情報:管理情報における計量指標を定める。

- **主題情報**
  移転される実情報、すなわちソフトウェア要求、設計、プログラム、テストに関する情報。これは、ソフトウェアに現れる概念を用いて記述されるが、多くの開発者やツールは、ある時点ではある特定の観点、またはビューからこの主題情報を眺めている。そして、観点が決まれば、その情報は何らかの方法で表現することができる。すなわち、主題情報はさらに以下のように分解される。

  - **概念情報 (concept information) と関係情報 (relationship information)**
    ソフトウェアをみる観点やモデルに依存せず、ソフトウェアが本来持つ概念要素と、それら要素間の関係を示す情報。P1175 では、概念情報として次のものが例示されている。
    * アクション
      変換、トランザクション、プロセスなど。
    * データ
      アクションの対象となる; 情報、データ関係など。
    * イベント
      アクションのタイミングと同期を制御する; 時刻印、割り込みなど。
    * 条件
      アクションに対する制約、制限、限界。
    * 状態
      アクションのコンタクトを与える
  - **局面情報 (perspective information)**
    実体関連、データ/制御フロー、状態遷移など、特定のビューに従う概念情報と関係情報の部分集合。
  - **表現情報 (presentation information)**
    局面情報を画面や紙の上、あるいはファイル上に表現する方法に関する情報。表現形式には、グラフィックス、テーブル、自然言語、データ交換のための専用言語 (例えば、CDIF[4](3.4.1 節)) などがある。例えば、グラフィックス表現では、シンボル、シンボル属性、シンボル構成、シンボル配置などが表現情報として記述される。

ここで示したソフトウェア意味情報、特に主題情報の分類は、ソフトウェアプロダクト本来の意味を、設計方法論や開発組織の方針、あるいは特定の計算機、オペレーティングシステム環境から分離する上で非常に優れている [42]。概念情報の要素がソフトウェアの

意味のすべてを構成する素となるので、その同定が重要な鍵となる。しかし、これに関して、次に示す問題点がある [54]。

- ソフトウェアにおけるすべての能動的要素 (ソフトウェアの動作する側面を捉える意味的情報要素で、最終的には計算機の命令実行の集合) をアクションだけで代表できるのか。
- 条件は、判断の対象が複数あり、また判断の種類を示す要素も必要であるから原子的ではない。
- 状態は局面によっては (例えば、プログラムでは)、データとして実現されるので、データと状態には重複がある。

このように、ソフトウェアの複雑な要素関係を整理することは、重要である。それは、ソフトウェアの意味的な要素の同定が難しく、また一連のプロダクトにおける意味要素の重複も多いからである。そのため、上で挙げた問題を解決し、ソフトウェアの意味情報として整理することは、リポジトリの機能を十分に発揮するためにも有用である。

3.4 既存のデータ統合技術：CDIF とPCTE

この節では、現在までに構築されたデータ統合技術として CDIF[4] と PCTE[3] を挙げる。これらの技術は、データ統合に貢献してきているが、未だ CASE ツール開発に広く使われてはいない。これは、本質的にデータ統合が難しいからである。それゆえ、XML を含めて、様々な手法を試して、有用性を調べることが、データ統合技術の進歩に必要である。

3.4.1 CDIF : CASE Data Interchange Format

CDIF は、CASE ツールやリポジトリの仕様ではなく、CASE ツール間のデータ交換に用いられる標準フォーマットである。ここでは、CDIF の概要とその問題点について述べる。

CDIF の概要

CDIF が扱うデータは主に上流工程で扱われる開発対象システムの仕様や構造の情報である。例えば、実体関連図やデータフロー図である。CDIF では、このような仕様図の意味情報と表示のための情報を区別して扱う。この区別により、必要な情報のみを CASE ツールの方で容易に選択できる。これらの特徴を実現するための基本構造は、次に示す 4 つの階層を基にして構成されている。

- **メタメタモデル**
  最上層は、メタメタモデルと呼ばれ、メタモデルの構造や意味、制約を明示的に定義している。メタメタモデルは、メタモデルの構成要素を構築、拡張するための規則を与える。このレベルの定義は CDIF において固定であり、転送対象とはならない。

- **メタモデル**
  メタメタモデルの一段下の階層は、メタモデル層である。CDIF では、意味の情報の転送に際し、サブジェクトエリア (Subject Area) という概念を導入している。これは、例えば、データフローモデル、あるいは実体関連モデルのような同一抽象度の仕様図式において、共通の意味が取れる領域のことである。CDIF では、モデル情報を記述する規則をサブジェクトエリアごとに定義しており、これをメタモデルと呼んでいる。つまり、データフロー図や実体関連図の記述規則 (例えば、各図式の構成要素と要素間の関係や属性) がメタモデルに規定されている。また、メタモデルでは、各サブジェクトエリアの意味情報と表示情報の定義方法が表記される。

  - **意味情報と意味モデル**
    意味情報は モデルを記述するために必要な本質的な情報である。例えば、データフロー図では、どんなプロセスがあって、そのプロセスからどのプロセスにどんなデータフローがあるか、などの概念要素の関係に関する情報である。意味モデルでは、仕様を構成する概念とその間の関係、例えば、データフロー図においては、データフローや、プロセス、ファイル、データの源泉/吸収という 4 つの概念とその間の関係を定義する。

  - **表示情報と表現モデル**
    表示情報は、CASE ツールが意味情報を表示するための情報である。例えば、データフロー図では、プロセスをどんな図形で表現して、大きさはどのくらいで、どの位置に、どんな色で表示するか、などの描画に関する情報である。表現モデルにおける図情報の構成において、描画項目 (Drawing Item) が描画の最小単位である。描画項目には、線や三角形、菱形、正方形、テキストなど 16 種類がある。描画項目では、単純な図しか表示できないので複数の描画項目を合成して表現する必要がある。

- **モデル**
  実体関連図やデータフロー図などの仕様図式で個々に記述された開発対象システムの仕様や構造の情報を示す。これは、CASE ツール間でやり取りされる転送データ、すなわち CDIF の主たる転送対象である。

- **データ**
  最下層は、実世界で扱われるデータを指す。これは、例えば、あるデータフロー図において記述される開発対象システムにおいて実際に扱われるデータそのものを指すものであり、CDIF の扱う範囲外である。

CDIF の問題点

CDIF は、1990 年代初頭より EIA[1]よって開発され、ISO(国際標準化機構) でも審議が続けられているが、プロダクトの多様性への対応に苦労している。特にメタモデルでは、各々に多くのバリエーションを持つモデルを次から次へと定義しなければならないことが問題である。例えば、データフローモデルや、状態とイベントを中心とするモデル、データベースを中心とする領域、オブジェクト指向分析・設計など、多くのモデルに対応しなければならないので、作業量が膨大になり開発コストが高くなる。

また、CDIF では下流工程への支援が遅れているため [44]、我々の目的には CDIF は適さない。また、XML への変換が試みられている [35]。

3.4.2 PCTE : Portable Common Tool Environments

PCTE はソフトウェアリポジトリの機能構成を明確に示した代表例である。ここでは、PCTE の概要とその問題点について述べる。

PCTE の概要

PCTE は、ソフトウェア開発をサポートする環境の一部として、そこで扱われるデータの処理機能群に対するプラットフォーム独立なアクセスを提供する [43]。つまり、PCTE にはデータ統合機能を提供するだけでなく、標準プラットフォームを提供することによりツールの可搬性を高める働きがある。PCTE に準拠したツール間であれば異なるベンダーの製品であっても、リポジトリに蓄積された情報を共有できる。

重要となる概念は、オブジェクトとリンク、属性、スキーマで、すなわち、実体関連モデルである。PCTE システムの中核となるのは、オブジェクトベースである。ここで言うオブジェクトとは、オブジェクト指向でいうところのオブジェクトではなく、既定の型として主に次に示す 5 つがある。しかし、これらのオブジェクトには、継承の概念はある。

- ファイル
- ボリューム
- デバイス
- パイプ
- メッセージキュー

オブジェクトベースのデータモデルは、オブジェクトが実体 (entity) であり、関係 (relationship) にあたるものが双対のリンクである。それらは、共に属性 (属性名と属性値) を

[1]Electronic Industries Association

持ち、さらにオブジェクトは、コンテンツ (contents) を持つこともできる。コンテンツは、そのオブジェクトの持つ実データ (バイト列) である。

これらの要素概念のうち、特にリンクに PCTE の特徴がある。リンクには、次に示すプロパティがある。

(O) 始点依存性
リンクの始点のオブジェクトに、このリンクの生成、削除を行なう権限が必要である。

(R) 参照完全性
リンクの終点のオブジェクトをできない。参照完全性をもつ逆リンクを持つ。

(E) 存在特性
リンクの終点のオブジェクトと、このリンクの存在が常に同期する。

(C) 構成特性
リンクの終点のオブジェクトが始点のオブジェクトの構成要素となる。

これらのプロパティの組み合わせによって、次に示すリンクのカテゴリが定められる。

COMPOSITION	(O), (R), (E), (C)
EXISTENCE	(O), (R), (E)
REFERENCE	(O), (R)
IMPLICIT	(R)
DESIGNATION	(O)

PCTE では、個々のオブジェクトが識別できるのは、構成特性を持つリンクの属性によるものである。そのため、PCTE ではリンクのカテゴリ、型、多重度、属性を中心として、リポジトリの定義、操作、管理を行なう。

リポジトリに格納するデータは、オブジェクト、リンク、属性のインスタンスである。これらの構成は、予め各々の型の集合、すなわちスキーマとして定義しなければならない。PCTE では、これをスキーマ定義集合と呼んでいる。1 つの型は複数のスキーマ定義集合に現れることができる。型を特徴付けるデータ (特性データ) には、属するスキーマ定義集合によって変えられるもの (外部特性) と変えられないもの (内部特性) がある。これは、型の再利用性を高めるのに有効で、スキーマ定義集合間での型の移入、移出という仕組みによって実際に利用する。PCTE では、次に示す型を各概念要素の型付けに用いている。

- **オブジェクト型**
  オブジェクト型自身の名前、持ち得る属性、流出リンク、流入リンクの型が外部特性である。一方、リポジトリ全体で一意の型識別子は内部属性である。

- **リンク型**
  複数のインスタンスを持ち得るリンク型の場合、一意にインスタンスを識別するキーとなる属性型は、オブジェクトでなくリンクが持つ。それによって、オブジェクト間の 1 対多の関係を表す (多重度多)。キーは複数の属性の組で決められてもよい。キー属性がない場合、そのリンク型はオブジェクト間の 1 対 1 関係を示すものとなる (多重度１)。リンク型については、型の名前とキー属性以外の属性に型が外部属性、型識別子、カテゴリ、多重度、キー属性は内部特性である。内部属性には、もう 1 つ安定性と呼ばれるものがある。これの特性が指定されると、そのリンクの終点オブジェクトの変更が禁止される。

- **関連型**
  リンク型は単独では、リポジトリに存在し得ず、関連型である。関連型は、双対をなす 2 つのリンク型の組で構成される。関連型を構成する 2 つの双対リンク型は、常に同時に生成、削除される。順方向、逆方向、各々のリンク型が関連型の内部特性となる。外部特性は順リンク型の始点 (逆リンク型の終点) や、順リンク型の終点 (逆リンク型の始点) となるオブジェクト型である。

- **属性型**
  属性値の型 (整数、文字列、心理値など) や、初期値、属性型がキー属性として適用されていることを示す特性が内部特性である。外部特性は型の名前や、この属性型が適用されるオブジェクト型、またはリンク型である。

PCTE の問題点

PCTE を基盤にすることで、その上で稼働するツールの扱うデータモデルを明確にし、共通化することができる。しかし、PCTE は、1992 年頃には製品も出回り、1995 年には国際規格 (翌年に JIS 規格) となったが、実用的には普及してない。それは、次に示す問題点が原因であると考えている。

- オブジェクトやリンクに付けられる属性名、属性値の意味的関係の管理は、リポジトリの中で統一的に管理しているのではなく、すべてツールに任されている。

- オブジェクトの粒度としては、ファイル単位の大きさを想定しているため、細粒度の設計情報の管理に関しては、性能上の問題が解決されていない。

- 同一抽象レベルの情報 (例えば、データフロー図と実体関連図) を一度に扱えることが望ましいが難しい。

3.5 XMLによるCASEツール開発支援

3.5.1 XMLの適用とその利点

XMLをCASEツールプラットフォームに適用することで、データ統合が実現できる。なぜなら、DTDを決めることで、共通フォーマットが規定でき、DTDに適合した(妥当な)XML文書であれば、ツール間でデータ交換ができるからである。また、多様なプロダクトに対応して、各々のDTDが定義でき、プロダクトそのものをXML文書化し格納できるからである。つまり、XMLを用いることで、3.3節の「データ統合へのアプローチとその位置付け」で示した**共通フォーマット**と**スキーマ定義**の特徴を有効に活用できるのである。これは、CASEツールプラットフォームにXMLを用いる大きな利点である。

本研究においては、特に下流CASEの統合化に注目している。それは、ソフトウェア開発の大部分がテストや保守に費やされていることや、盛んに研究が進められている上流工程の成果を支援できる下流工程の開発環境が整っていないことが問題となっているからである。XMLをCASEツール開発に適用することで、各々のCASEツールに必要であった対象ソフトウェアの解析器の部分を共通データとして細粒度で規定する。これにより、CASEツール開発のコストを削減することが狙いである。また、それまでは内部データとして扱ってきた解析器の出力するデータが、開放的になるので、これを基にツール間の連携を容易にとることができる。

この下流CASEの統合化に実現するために、特に有効に活用できる利点がXMLを用いることにはある。それは、上で述べたCASEツールプラットフォームとして利点に加えて、主に次の利点である。

- XMLはリッチなデータ構造(基本的には木構造、他は例えば、グラフ構造やテーブル構造)を提供するため、データ統合に不可欠なプロダクトのスキーマ定義をXMLのタグとして簡潔に定義できる。
  - プログラム構文構造は、XMLのエレメントの入れ子構造として自然に表現できる。
  - プログラム要素間の関係(例えば、変数の定義、参照関係)をID/IDREFリンクとして表現できる。この表現は簡潔で扱いやすい。
  - シンボル情報は、XMLの空エレメントとしてリニアに表現できる。
- 妥当なXML文書として扱われないが、整形式なXML文書として表現すれば、不完全なデータ(例えば、バグのあるコード)でさえも扱うことができる。
- XMI(UML[2]図のXML表現形式)[6]の標準化が進んでいる。下流CASEツールにXMLを適用すれば、上流と下流の間のデータ統合が期待できる。XMIについては、3.5.2節で述べる。

[2]Unified Modeling Language

- XML Namespace[13] を使うことで、XMI のような他のマークアップ言語との、相互運用が可能可能になり、多様なプロダクトを扱える。

上では、下流 CASE に特化した利点を挙げたが、XML の特徴で CASE ツール開発に活用できる利点は、これだけではない。次に挙げる XML の一般的な特長は、CASE ツール開発においても有効である。

- XML は、構造化データに加えてプレーンテキストとしての利点も持つ。そのため、人が読むことができるし、**sed** や **grep**、**perl** のようなテキスト処理ツールも XML 文書に適用できる。
- テキストデータは、改行コードの問題を除けば、バイト順のようなデータ変換の際に発生する問題とは無縁なため、異なったプラットフォーム間で XML 文書を容易に交換できる。
- XML パーサや、XSLT[11]、DOM[10] など、XML 文書を柔軟に処理する標準的なツールが多く存在する。
- Java を始め、多くの汎用プログラミング言語で操作が可能で、Web ブラウザなど、既存のユーザインタフェースが活用できる。

このように、CASE ツール開発に XML を用いる利点は多くある。XML は、DTD をうまく設計することで、CASE ツール間でのデータ共有を実現することができる。DTD の設計において、多種多様なモデルの持つ数多くの意味要素をタグやその中の属性を使って、どのように表現するかが難しいことである。しかし、その DTD でプロダクトを適切に定義できれば、プロダクトを XML 文書に変換するトランスレータを用意するだけで、CASE ツールプラットフォームの役割を担うことができる。

3.5.2 上流工程を支援する XMI

現在、XML によるソフトウェア開発の上流工程を支援する枠組みは確立されつつある。XMI[6] は、XML による UML のテキスト表現形式で、OMG[3]によって標準化が進められている。

XMI は、ツール間やデータベース間、アプリケーション間の UML 情報交換に役立つ。例えば、XMI によって、１つのビジュアル・モデリング・ツールから、他の設計ツール、アプリケーション、データベースへの UML 図の交換を可能にする。また、１つの開発ツールに縛られることなく、選択肢が拡大する。これにより、上流 CASE の開発コストが削減できる。

[3]Object Management Group

実際にXMIをサポートするCASEツールも開発が進んでいる。例えば、Rational Rose[7]には、作成したUML図をXMIに従ったXML 文書に変換できる。

しかし、XMLによる下流CASEの統合に関する研究はまだ少ない。下流CASEの統合が実現すれば、上流のモデリングと下流のソースプログラムの双方向変換が可能になる。つまり、ソフトウェア開発の上流部と下流部間の大きなギャップをXML関連技術を使うことで埋めることができる。我々が提案する「XMLによる下流CASEのデータ統合」のゴールの1つは、この上流と下流の協調に役立つことである。

第4章　XMLを用いたCASEツールプラットフォームの提案・実現

4.1　XMLを用いたCASEツールプラットフォームの提案

我々がゴールと考えるCASEツールプラットフォーム (図4.1) では、多種多様なプロダクトをXML文書に変換する。例えば、仕様図面がUML図ならXMIに、ソースコードがANSI Cプログラムなら ACML(4.2節) に従って変換する。

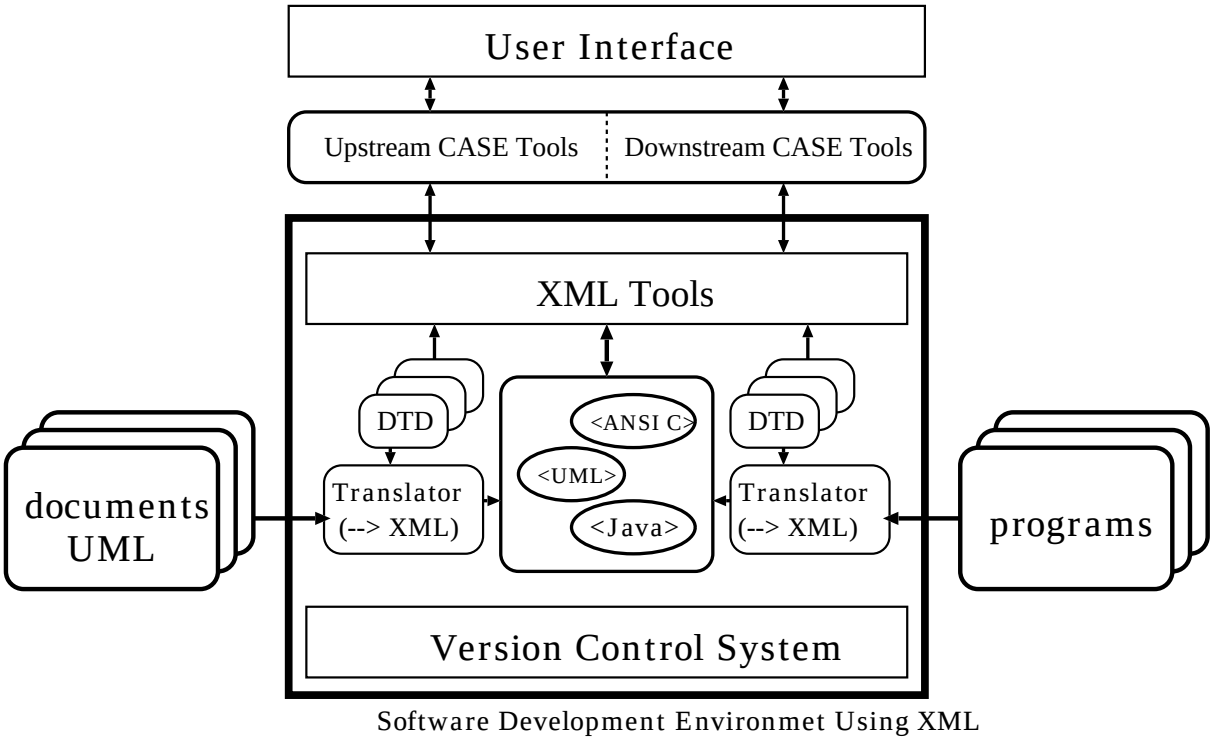

図 4.1: XMLを用いたCASEツールプラットフォーム

XML文書化したプロダクトは、既存のXML関連技術を通して、このツールプラットフォーム上で開発したCASEツールを活用して作成や変更、解析を行なう。このプロダクトのXML文書化によって、プロダクト間の一貫性を維持すると共にCASEツールプラットフォームと各ツールの間、さらには各ツール間に開放性を与えることを図る。つまり、

これはソフトウェアの分析・設計から、テスト・デバッグまでを一貫して支援できるソフトウェアシステムである。

このXMLを用いたCASEツールプラットフォームでは、次に挙げる機能の実現が期待できる。

- ソースプログラムに対して制限 (例えば、言語の種類やプログラムの完成度) を与えず解析ができる。
- ソースプログラムに表れるプログラマの意図 (例えば、デザインパターンの検出) が理解できる。
- プロダクトの意味 (例えば、プログラムならスコープ規則 [58]) を考慮に入れた情報管理ができる。
- 上流CASEと下流CASEの協調作業 (例えば、UML図とソースプログラム間の整合性検査) ができる。

これらの機能の実現には、XMLの特徴 (3.5.1節) を有効に活用することが重要である。以下では、 これらの機能の詳細について述べ、XMLを用いる場合についての問題点を示す。

4.1.1 多種多様なソースプログラムの解析

対象ソフトウェアについては、種々のプログラミング言語に対応し、各々、既存のプログラム、エラーを含んだプログラム、作成中のプログラムを扱えることを目指す。

既存の統合開発環境、いわゆるIDE[1](例えば、VisualC++) は、特定のプログラミング言語をターゲットにしている。多くのプログラミング言語 (例えば、CやC++、Java、ML) が存在する今日では、プロダクトに応じて、適切なプログラミング言語を選択し、コーディングを行なう。そのため、プログラミング言語を限定した開発環境では、多様なプロダクトへのサポートが不十分になる。そこで、我々が提案するXMLを用いたCASEツールプラットフォームでは、種々のプログラミング言語に対応することも目標にしている。

この対象プログラミング言語の多様化の他に、あらゆる状態 (例えば、バグの有無や完成度) のプログラムへの対応も目標にしている。ソフトウェアの開発、保守には、既存のプログラムに対する解析は不可欠である。我々は、既存のプログラムをXML文書化して、ソースコードの理解やある特定の変数名の変更、コードの最適化、リバースエンジニアリングなどに役立てたいと考えている。

仮にその解析するプログラムにエラーがあったとしても、そのプログラムは、XML文書として処理できるようにするべきである。なぜなら、エラーを含んだプログラムにも、

[1]Integrated Development Environment

デバッグやテストのために価値のある情報を得ることができるからである。また、作成中のプログラムに対しても同様である。この XML 文書化により、作成中のプログラムに対してもレビューを行なうことができるようになり、早期に設計やコーディングに修正、変更を行なえる。

ここで問題になるのが、DTD の設計である。正しく動くプログラムを対象とする場合、その言語の構文規則を基に DTD を設計するが粒度が問題になる (4.2.2 節)。また、エラーを含むプログラムや作成中のプログラムに対しては、XML 文書として、構文エラーや静的エラー (例えば、宣言してない変数の使用)、動的エラー (例えば、メモリリーク)、あるいは、完結していない状態をどのように表現するかが問題である。

4.1.2 プログラマの意図の理解

ソースプログラムには、コンパイラが解釈できないコメントや、コーディング規則、デザインパターンなどの”プログラマの意図”が含まれている。この”プログラマの意図”を理解することは、そのソースプログラム中の要素間の関係や仕様に対するプログラマの指針、テスト、保守に対する留意点などを把握するのに有効である。

簡単な例として、C プログラム・チェッカ Lint[30] を挙げる。lint プログラムの機能の 1 つは、制御フローとデータフローの解析を行ない、到達しないコードを検出する。しかし、/*NOTREACHED*/指示文をコメントの中に記述すると、到達しないコードがあるという警告を抑えることができる。例えば、次に示すプログラムでは、到達しない’case 2 :’ の部分にこの指示文を記述している。

```
int foo(void)
{
  int sw=1;
  switch (sw) {
    case 1 : puts("Reached case1");
             break;
    case 2 : puts("Reached case2");        /* 到達不可能なコード */
             /* NOTREACHED */              /* 警告を抑制*/
             break;
  }
  return (0);
}
```

lint プログラムでは、このような簡単な意図は、/* NOTREACHED */を始めとする幾つかの指示文を記述することで表現できる。

上の例では、コメントの中で明示的に意図を表現していたが、実際は、コード自体に意図を含める場合もある。例えば、次の ANSI C プログラムの断片は、正しい外部宣言である。

```
extern int x = 999;                /* 正しい宣言 */
```

'=999' のような初期化子を伴った外部宣言は、定義として扱われる。しかし、これは全く正しい宣言であるが、他のプログラマやレビューアは、このプログラムに違和感を感じ、何らかの警告ととらえるだろう。多くのプログラマは、時々、意図して何らかの普通でないコーディングをすることで、他のプログラマやレビューに対して警告を促すことがある。しかし、どのようにして、このようなコード自体に含まれる意図を表現するかが問題である。

より複雑な意図になると、例えば、UNIX コマンド man は、環境変数 PAGER を経由して他の UNIX コマンドと関係を持つ。その際、PAGER にはテキストファイルの中身をユーザに見やすく表示するコマンドがセットされることを期待する。しかし、プログラムの中で、そのような漠然とした意図を記述することは困難である。

また、多くのプログラマは、コーディング規則 (例えば、ネーミング規則や暗黙のキャスティングの禁止) や、幾つかのデザインパターンを基にしてプログラムを作成する。コーディング規則は、ソースプログラムの可読性を向上させ、他のプログラマやレビューアが容易に理解できるようにする効果がある。これは、そのソースプログラムの読み手に対するプログラマの期待を表している。このコーディング規則を反映しているソースプログラムをどのように表現するかが問題である。

デザインパターンに関しても同様である。デザインパターンは、ソフトウェアやシステム設計を再利用するための技術であり、プログラミングにおける定石をカタログ化したものである。デザインパターンに沿ってプログラミングを行なうことは、ソフトウェア開発の効率化において、有用なものとなっている。これは、プログラマの設計ポリシーを反映するものである。しかし、デザインパターンが適用されている部分をソースプログラムから検出することは困難である。

このように、ソースプログラムに込められた他のプログラマやレビューアに対する警告や期待、設計ポリシーなどの”プログラマの意図”は、機械処理可能なフォーマットで表現することが難しい場合があり、XML 文書としてどのように表現するか検討しなければならない。

4.1.3 意味情報の管理・操作

XML で表現したプロダクトは、そのプロダクトの意味 (例えば、プログラムならスコープ規則 [58]) を考慮に入れた情報管理が期待できる。

一般にソフトウェア開発において、プロダクトに関する情報の管理は必要不可欠なものである。この情報の管理をするための代表的な CASE ツールが、バージョン管理システム (例えば、CVS[2]) である。バージョン管理システムは、ファイルや種々のデータの内容を修正したり変更したりする場合に、その修正及び変更内容を履歴として保存するように設計されたソフトウェアである。これは、ソフトウェア開発の全工程にわたって支援を行う重要なものであり、XML を用いた CASE ツールプラットフォームにおいても同様である。しかし、既存のバージョン管理ツールでは、意味のある変更と、意図せずに発生した意味のない変更 (例えば、空白やタブ文字の変化) とを区別できない。

そこで、XML の構造を認識するバージョン管理システムが必要になる。XML の構造が認識できれば、XML 文書化したプロダクトは、意味を保持したまま管理できる。これにより、例えば、プログラムのデバッグの際、このバージョン管理システムを使って最後の変更と意味解析後のレベルで比較ができるので、より的確なデバッグ作業が可能になる。また、多数のプログラマとの協調作業によるソフトウェア開発の際、あるプログラマが特定の変数についての変更を行なうときに、その変数がいつ、誰によって定義されたものか分かるので、大規模ソフトウェア開発において効率的な修正、変更作業が可能になる。

しかし、XML 文書が意味的に等価であるか否かを判別することは、困難である。その主な原因 1 つは、ID 型の属性に関するものである。ID 型の属性は、各々のエレメントに対し、ユニークに与えられる識別子である。IDREF 型の属性と組み合わせて用いることで、各エレメント間にリンクを張ることができる。しかし、この ID 値に変更を加えた場合、異なったバージョンで同じ ID 値を共有できなくなり、それまでのリンクが切れ、意図しない変更が行なわれてしまう。また、同じ木構造で同じリンク関係を表現する XML 文書であっても、ID 値のみがすべて変化してしまうと、まったく異なった XML 文書と解釈されてしまう。このように、XML を基にしたバージョン管理には、複雑さが伴い困難である。

4.1.4 上流・下流の統合

我々の目標とする「XML による下流 CASE のデータ統合」は、上流 CASE と下流 CASE の統合化を実現するに際して重要な役割を担う。なぜなら、3.5.2 節で示した XMI による上流 CASE の統合化が確立されようとしているからである。これら上流 CASE の統合と下流 CASE の統合が実現すれば、上流プロダクトと下流プロダクトを XML 文書化し、各々のプロダクトの構成要素間に関連付けをすることで、上流 CASE と下流 CASE の統合化、さらに協調作業が可能になると期待できる。

例えば、既存のソースプログラムからモジュールの関連や上述したデザインパターンの検出をすることで、そのプログラムの設計情報 (例えば、分析モデルや設計モデル) を生成することができる。また、仕様図からソースプログラムまで関連付けることで、上流工程から下流工程の間のどこかで、修正、変更作業を行なった場合、その修正や変更が影響

[2]Concurrent Versions System

を及ぼす部分も同時に更新することができる。コーディングの際に設計上の誤りや考慮していなかったことを発見した場合、修正、変更作業が容易になる。

これにより、例えば、設計結果から自動的にプログラムを生成する CASE ツールや、リバースエンジニアリングを支援する CASE ツールなど、上流と下流を通して支援する CASE ツールを開発することもできる。

しかし、問題は、上流 CASE と下流 CASE の対応づけをどのようにするかである。つまり、共に XML というフォーマットで表現できるが DTD は異なるかもしれない。この異なる DTD で表現された XML 文書を、どのようにして関連付けるか検討しなければならない。

4.2 実現したCASEツールプラットフォームの概要

4.1 節で述べた CASE ツールプラットフォームは、大規模なため、一度にすべてを実現するのは難しい。そこで、本研究では XML を用いたシフトウェア開発環境を構築する第一歩として、ANSI C プログラムのみをターゲットにした CASE ツールプラットフォームを実現した (図 4.2)。

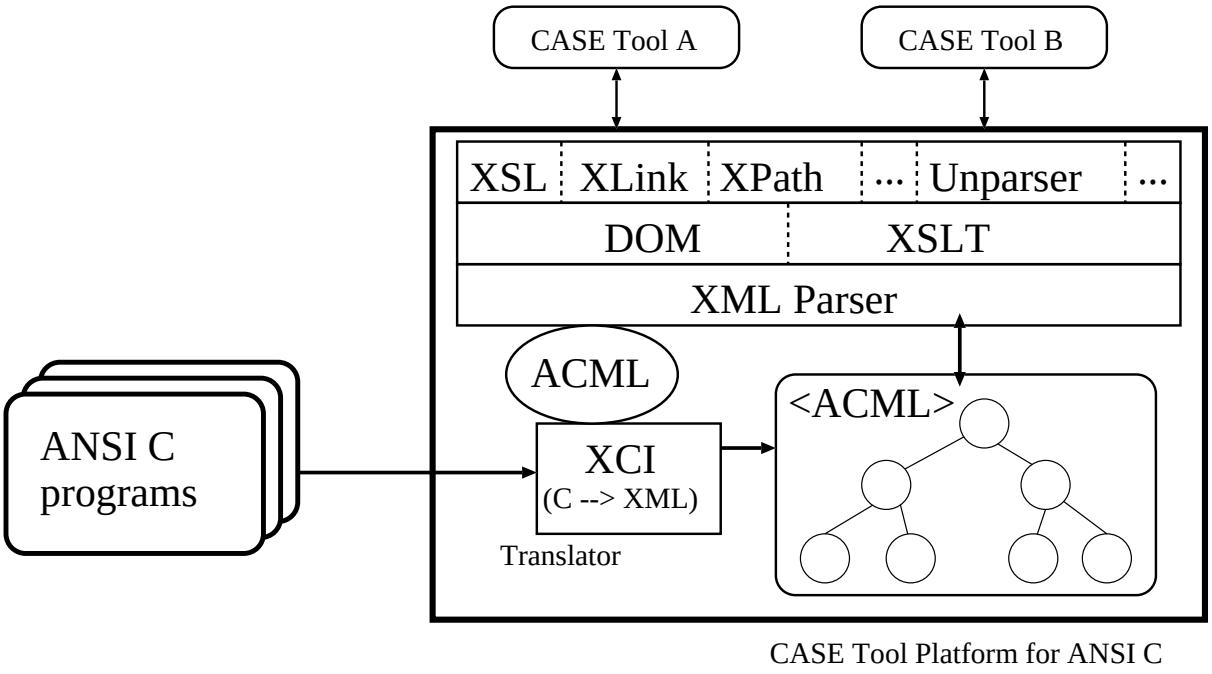

図 4.2: ANSI C のための CASE ツールプラットフォーム

XML が実用的に活用できるか確認するためには、我々は言語のサブセットではなく、フルセットをサポートすべきと考えている。我々のシステムは、ほぼフルセット[3]の ANSI

[3]現在は前処理命令 (例えば、`#include`) やライブラリ関数 (例えば、`signal`)、システムコール (例えば、

C をサポートしている。

実現した CASE ツールプラットフォームの構成要素は、次の 3 つである。

- ANSI C 用に定義した DTD(ACML と呼ぶ)
- ANSI C プログラムをタグ付けされた XML 文書に変換するトランスレータ XCI(4.2.3 節)
- アンパーサや、応用例として、プログラムスライシングツールとクロスリファレンサ

図 4.2 は、これら 3 つの連携を示す。まず、ANSI C プログラムは、XCI によってタグ付けをし、ACML 文書に変換する。その ACML 文書は XML パーサを通じて DOM や XSLT で、プログラムスライスを抽出したり、ソースコードを表示したりするために処理する。

4.2.1 ANSI C を対象にする理由

本研究において、我々は ANSI C のみをターゲットにするのは、4.2 節の始め述べた通り、ターゲット言語を 1 つに絞ることが、我々のゴールに向かう第一歩となると考えたからである。では、なぜ、その 1 つを ANSI C にしたのかというと、理由は主に次の 2 つある。

1. ANSI C は、標準的なプログラミング言語であり、多くのソフトウェアで使われている言語である。
2. 特定のコンパイラ (例えば、GCC) を対象にすると、独自拡張した文法 (例えば、**asm** 構文) を考慮に入れなければならず、それは複雑でサポートするのに困難が生じる。
3. 一般に C 言語は、その精神と構造的な問題から、バグが発生しやすく、CASE ツールによるテストや保守が必要である。

以下では、上記の 3 項目についての詳細を述べる。

汎用的な ANSI C

C 言語は、UNIX システム上で開発された汎用プログラミング言語である。コンパイラ、OS、その他のソフトウェアは、ほとんど C で書け、UNIX 上で開発されたものに関してはオープンソースであることから、C は広く用いられるようになった。しかし、多様な計算機環境に対応するために、コンパイラ開発者は C を各々拡張する方針をとった。その結果として仕様の異なる C が多く存在するようになり、C 言語に対して標準規格が必要になった。

`fork`) をサポートしてない。詳細は 4.2.3 節で述べる。

そこで、アメリカ国内標準協会 (ANSI[4]) は、1983 年に C に関する現代的で総括的な定義を行なう目的の委員会を発足させ、1988 年に”ANSI C”が承認された。これにより、それまで各々異なっていた文法、前処理やライブラリ関数が統一化され、ANSI C は、それ以前の C に比べ、移植性、互換性が高く、より広範囲で用いられる汎用プログラミング言語になった。

独自拡張による複雑化

多くの ANSI C の標準規格に準拠したコンパイラは、ANSI C の構文規則に対し、独自に拡張した文法を加え、特殊な関数や機能を提供している。そのため、現在のところ、特定のコンパイラの拡張文法まで考慮に入れた C 言語は、ターゲット言語として適切ではないと考えている。なぜなら、その独自拡張により、対象とする文法が複雑になり、解析をより困難にするからである。

例えば、GCC における独自拡張を幾つか挙げると、インラインアセンブラのための **asm** 構文や、インライン展開のための **inline** 宣言、式の型を取得するための **typeof** 演算子などがある。これらは、ANSI C 標準規格にはない機能で、強力で有用なものであるが、あまりに多くの拡張があり、GCC の文法を複雑する要因となっている。

ANSI C の性質

ANSI の目的は、次に挙げる C の精神 [56] を保持しながら、標準規格をまとめることにあった。

- プログラマを信頼する
- プログラマの必要と判断したことをするのを妨げない
- 言語仕様を小さく、単純にとどめる
- 1 つの操作には 1 つの手段しか提供しない
- 移植性を損なっても、速くなる方向を選ぶ

これにより、ANSI C は、比較的に言語構造が単純で、低レベルな処理が可能であるという長所を得た。しかし、その反面で、その言語構造の単純さが、複雑なプログラムの理解を困難にしたり、その低レベルな処理が、OS の暴走やファイルの破壊の原因になったりする。つまり、ANSI C はプログラマに強力な機能を提供する反面、多くの予期せぬエラーを招くことになる。

例えば、ポインタ操作の誤りによるメモリリークやメモリ破壊、メモリの二重使用といったエラーは致命的な現象を起こすことが多い上にデバッグが容易でない。また、ANSI

[4]American National Standards Institute

C は、強く型付けられた言語 [58] でない上に暗黙の型変換も許すことから、プログラム実行中に型エラーが生じ、プログラマの意図と反した処理がされる可能性もある。

よって、ANSI C で書くソフトウェアには、他のプログラミング言語 (例えば、Java や Standard ML) 以上に、テストや保守に多大なコストをかけざるを得ない。このため、CASE ツールによるサポートは、必要不可欠なものになっている。

4.2.2 ACML : ANSI C Markup Language

ACML は、我々が開発した ANSI C 用マークアップ言語で、データ統合のためのスキーマである。ACML の DTD は、ANSI C プログラムにタグを付け、ACML 文書を生成する。ACML の DTD は付録 A に示す。

ACML は、抽象構文木の他に、シンボルや型、参照・定義の間の関係、制御フローなどの静的意味に関する情報を表現し提供する。このため、ACML は静的プログラムスライシングツールやクロスリファレンサ、静的テストケース生成器のような静的にプログラムを解析する CASE ツール開発で力を発揮できることが期待できる。実際に ACML は、プログラムスライシングツールとクロスリファレンサの実験的実現において有用であることを第 5 章で示す。

この節では、DTD 設計の概観を示し、ACML はどのように ANSI C プログラムを表現してるかについて述べる。

ANSI C のための DTD 設計

ANSI C プログラムを解析するためには、そのプログラムに対するモデル化が必要である。ANSI C のための DTD、つまり、ACML を設計する際に、次に挙げる 2 点が問題になった。

1. どのような種類の情報を ACML として提供するべきか。
2. 対象とするプログラムを、どの程度の粒度で ACML としてモデル化するべきか。

本研究では、”下流 CASE の統合”の第一歩として、下流工程で用いる CASE ツール開発に対応するため、この種の解析に最低限必要な静的意味情報を細粒度でモデル化する ACML を設計した。以下では、静的意味情報を含め、ACML が提供できる情報を挙げ、静的意味情報を細粒度でモデル化したことについて述べる。

ACML が提供できる情報　ほとんどの下流 CASE では、構文構造に関する情報が必要なため、ACML に含める情報として静的意味情報は最も基本的で重要である。この静的意味情報を含め、ACML として提供したい情報には幾つか候補がある。それは、主に次の 5 種類である。

- **静的意味情報**
  抽象構文木の他に、型の構造やシンボル、制御フローに関する情報を提供する。

- **動的意味情報**
  例えば、変数値の履歴と他のプロセスや OS との通信に関する情報を提供する。

- **プログラミングスタイルを表す情報**
  例えば、字句情報や設計ポリシーに関する情報を提供する。

- **メンテナンスのための情報**
  例えば、テストのための指示書やデータ、デバックの進捗状況を提供する。

- **解析結果を表す情報**
  上記以外の派生データ (例えば、プログラムスライス) を提供する。

これらのプログラムに関する情報を DTD として、どうのようにして、うまく設計するかは難しいことである。**動的意味情報**に関しては、特に、コンパクトで柔軟、かつ統一的に設計することは難しいことである。これについては、現在も活発な研究が行なわれている [39] が、未処理のトレースが、あまりに大きく、あまりに非構造的になりやすいことが、大きな問題となっている。

また、**プログラミングスタイルを表す情報**には、ソースプログラムの表示のための字句情報 (例えば、空白やコメント、インデント) と、プログラマの意志を示すコーディングのスタイル (例えば、変数や関数の名前の付け方) やルール (例えば、暗黙の型変換の禁止)、設計ポリシー (例えば、デザインパターンの使用) に関する情報がある。このプログラミングスタイル [55] の中でも、特に、プログラマの意志を示す情報については、機械処理が可能なフォーマットで表現することが難しい場合がある。

メンテナンスのための情報と**解析結果を表す情報**については、その元となる情報 (例えば、テストヘッドやバグレポート) の整備や解析を行なう CASE ツール (例えば、プログラムスライシングツール) の実現が必要である。

このように、ACML には、種々の情報を提供できると我々は考えている。いずれも有用な情報であるが、明らかにファイルサイズと計算時間の間にトレードオフがある。このため、ACML に含めるべき情報については、十分な議論が必要である。現時点では、その中で最も基本的、かつ重要である静的意味に関する情報のみを ACML として表現している。また、異なったプログラミング言語や異なった CASE ツールで利用できることも考慮に入れるべきである。

プログラムモデルの粒度　我々は、ACML の設計の際、細粒度のアプローチを取った。つまり、ACML によるタグ付けの対象を関数やステートメントだけでなく、リテラルや変数を含め、全ての言語構成要素を考慮に入れたのである。なぜなら、多くの高度なプログラムスライシングツールやクロスリファレンサは、詳細な構文情報を必要とするからである。この細粒度のアプローチの結果として、ACML に従ってタグ付けした ANSI C プロ

グラムは、非常に大きくなる傾向にある。表 4.1 で示すように、`#include <stdio.h>` を使った'Hello World' プログラムでさえも、ACML 文書に変換すると、そのサイズは 500K バイトを越える。なぜなら、ヘッダーファイルである `stdio.h` の中で、多数の宣言や型 (例えば、`struct` 定義や `typedef` ネーム)、識別子を導入するためである。

	ANSI C プログラム	ACML 文書
HelloWorld (`#include <stdio.h>` なし)	85 bytes	19,571 bytes
HelloWorld (`#include <stdio.h>` あり)	71 bytes	547,617 bytes
xml.c in XCI [5]	54,375 bytes	3,581,618 bytes
global.c in GNU Global [33]	27,908 bytes	3,202,088 bytes

表 4.1: ANSI C プログラムと ACML 文書のサイズ

この DTD のエレメントの粒度には議論が必要である。ACML は細粒度であり、例えば、識別子も 1 つのエレメントとして表現する。その一方、Badros[2] は、Java 用のマークアップ言語として JavaML[5](7.2 節) を提案した。Badros は、パースする際に用いる文法は、冗長すぎるので適切でないと主張した。Badros の目標の 1 つは、オブジェクト指向プログラミング言語で、共通に使われるマークアップ言語を開発することである。

多種多様なソフトウェアプロダクトに応じて、適切な粒度は異なる。上流工程での種々のプログラミング言語をまとめようとする要求には、JavaML のような粒度の粗い DTD が有用であるが、下流工程では、ACML のような細粒度の DTD が有用である。よって、我々は、特に下流 CASE の統合に注目しているので、細粒度の DTD である ACML を設計したのである。

ACML が表現する ANSI C プログラム

ACML は、ANSI C の構文と型、シンボル、制御フローに関する情報を表現している。以下で、各々の情報について XML を用いてどのように表現しているかを具体例を挙げて示す。

ANSI C の構文情報　ANSI C プログラムの構文構造は、DTD を使うことで、XML のエレメントの入れ子構造として自然に表現できる。DTD は、直感的には拡張 BNF[6] 記法に属性を与えたものである。そのため、ACML は、次の点から ANSI C 構文規則 [52] を、より簡単に表現できる。

- DTD で、拡張 BNF 記法を使用する。例えば、0 回、または 1 回の出現を示す'?' や、0 回以上の出現を示す'*'、1 回以上の出現を示す'+' などの記号の組み合わせで ANSI C 構文規則を表現する。

[5]Java Markup Language
[6]Backus Naur Form

- 例えば、ANSI C 構文規則の' 式' の表現は、優先度と結合性を付けるために 18 種類の異なる非終端記号 (例えば、$primary - expression$ や $additive - expression$) からなり、構文規則を複雑にしている。これを 1 種類の<expression>エレメントを使って単一化する。

- 抽象構文木にしたとき、同じ種類、または同じ数の子供を持つ、異なったプロダクションルールを区別するために、'rhs' 属性を導入する。例えば、上の' 式' の例で元々あった 18 種類の非終端記号は、この'rhs' 属性で表現する。

表 4.2 は、ACML の表現によって、プロダクションルールの数と非終端記号の数が減少したことを示す。その一方で、静的解析を基にする CASE ツール、特に静的プログムス

	プロダクションルール	非終端記号
ANSI C 構文規則	183	65
ACML	45	17

表 4.2: プロダクションルール数と非終端記号数の比較

ライシングツールやクロスリファレンサが必要とする構文情報を確保している。つまり、ACML は ANSI C 構文規則と同じ構造を表現したまま、複雑な ANSI C 構文規則を簡単にしている。例として、次の関数を挙げる。

```
int foo (int a, int b)
{
  if (a < b)
        return a;
  else
        return b;
}
```

これは XCI により、次の ACML 文書へ変換される[7]。

```
1  <function_definition id="X8086760">
2   <int/>
3   <declarator rhs="pointer_null">
4    <declarator rhs="func_new">
5     <declarator rhs="id">
6      <identifier rhs="identifier" ref="X8086760"> foo </></>
```

```
 7    <parameter_declaration rhs="dec">
 8    <int/>
 9     <declarator rhs="pointer_null" id="X8084e28">
10      <declarator rhs="id">
11       <identifier rhs="identifier"> a </></></></></>
12    <parameter_declaration rhs="dec">
13    <int/>
14     <declarator rhs="pointer_null" id="X8085078">
15      <declarator rhs="id">
16       <identifier rhs="identifier"> b </></></></></></></></>
17   <statement rhs="compound">

  <!-- if-else 文 -->
18    <statement rhs="if-else">

    <!-- 条件分岐部 -->
19     <expression rhs="less">
20      <expression rhs="identifier">
21       <identifier rhs="identifier" ref="X8084e28"> a </></>
22      <expression rhs="identifier">
23       <identifier rhs="identifier" ref="X8085078"> b </></></></>

    <!-- THEN 部 -->
24     <statement rhs="return">
25      <expression rhs="identifier">
26       <identifier rhs="identifier" ref="X8084e28"> a </></></></>

    <!-- ELSE 部 -->
27     <statement rhs="return">
28      <expression rhs="identifier">
29       <identifier rhs="identifier" ref="X8085078"> b </></></></></></></>
30  </>
```

上の ACML 文書は、ANSI C 構文を XML エレメントの入れ子構造として表現したことを示す。例えば、18 行目以降が if-else 文を示している。この 18 行目の<statement>エレメントには、3 つの子エレメントがあり、各々、19 行目の<expression>エレメント以下

[7]紙面の都合により、属性については、'rhs' と一部の'id'、'idref' 以外は省略した。また、全ての閉めタグは、</> に短縮した。

が条件式部を、24 行目の<statement>エレメント以下が THEN 部、27 行目の<statement>エレメント以下が ELSE 部を示している。また、プログラム間の要素関係は、ID/IDREF 属性を使って表現する。例えば、21 行目の<indentifier>エレメントの'ref' 属性が、'a' の定義部である 9 行目の<declarator>エレメントを指している。

このように、ACML は、ANSI C の構文規則を簡潔に表現し、同時に構文構造を、それと等価に表現できる。上の例で示した通り ACML は ANSI C プログラムを抽象構文木として比較的容易に表現する。

ANSI C の型情報　ANSI C プログラムの型については、抽象構文木にすると表現が異なるが、意味的には等価なものがある。例えば、次の 8 つの型の表現は、コードは異なっているが、どれも構文規則に従い、意味的には同じ型を示す。

(1) `unsigned long int`　(2) `unsigned int long`
(3) `long unsigned int`　(4) `long int unsigned`
(5) `int unsigned long`　(6) `int long unsigned`
(7) `unsigned long`　(8) `long unsigned`

ここで問題になるのが、上記の 8 通りの表現に対して、XML でユニーク (unique) に表現すべきか否かである。この表現は、ソフトウェアプロダクトによっては、必要となる情報である。例えば、'`unsigned long`' と'`int long unsigned`' との区別が必要なコーディングスタイル (coding style) を重視しなければならない場合に重要な情報となる。この区別は、異なるコードを対象としたユニークな表現でなければ、できないことである。

しかし、一般には、実際の型の意味に対してユニークな表現を用いる。なぜなら、型の同値性 [58] に関する検査に適しているからである。例えば、、演算子の型を同定したり、代入文の両辺の型が適合しているかどうかを調べたりすることが容易にできる。

これらを踏まえ、我々は、できる限り多様なプロダクトに対応したいので、ANSI C プログラムの型に関する表現を、ACML において 2 通り持つことで、上述の問題を解決した。例として、次の 2 つの宣言を挙げる。

```
unsigned long x;
int long unsigned y;
```

1 つ目の表現は、ACML が表現する抽象構文木の一部として、次の ACML 文書になる。

```
<unsigned/><long/>                    <!-- for x -->
<int/><long/><unsigned/>              <!-- for y -->
```

2つ目の表現[8]は、各々、同じ型の構造を示す次の ACML 文書になる。

```
<type>
  <t_prim>
   <t_int is_longlong="false" is_long="true" is_short="false"
          is_signed="false" is_unsigned="true"/>
  </></>
```

これにより、意味的な面では、上記の 2 つ目の ACML 文書で示した 1 種類の型として表現している。また、同時に 2 つの型の表現 ('unsigned long' と 'int long unsigned') に対して、ユニークな表現を持つことにもなる。それは、上記の 1 つ目の ACML 文書から、各々ACML の抽象構文木の一部を形成し、この 2 つを区別するための十分な情報を得ることができるからである。

正しいプログラムのみを前提にするなら、<t_int>エレメントの属性の、例えば、'is_long' と'is_short' は、一見すると不要な属性に見えるが、ACML において、意味的に誤った宣言 (例えば、'long short z;') も表現するために用意している。

一般に、型の宣言の中にポインタがあり、そのポインタが自分自身の型を指しているとき表現が複雑になる。例えば、この再帰的な型宣言 [58] はリスト構造で現れる。この場合、型を有向グラフで表現する必要がある。XML による型の表現では、この再帰構造を ID/IDREF 属性を使うことで表現する。例として、次のリスト構造を挙げる。

```
struct list {
   int          data;
   struct list *next;
};
```

これは、次の ACML 文書[9]になる。

```
<type id="X2618d70">                              <!-- ID -->
  <t_struct tag="list" num="2">
    <t_field name="data">
      <type>
        <t_prim>
           <t_int/>
```

[8] <type>エレメントの属性は省略する

[9] 一部の<type>エレメントの'id' 属性と'type_ref' 属性、<t_struct>エレメントの'tag' 属性と'num' 属性、<t_field>エレメントの'name' 属性以外は省略する。

```
      </></></>
    <t_field name="next">
      <type>
        <t_pointer>
          <type type_ref="X2618d70">          <!-- IDREF -->
            <t_empty/>
          </></></></></></></>
```

上の<type>の属性に注目すると、ID/IDREF を使ってリンクを張り、これが有向グラフの辺に値する。再帰的な型宣言の場合、この有向グラフにサイクルが現れる。(図 4.3) は、上の `struct list` のグラフ構造を表現したものである。

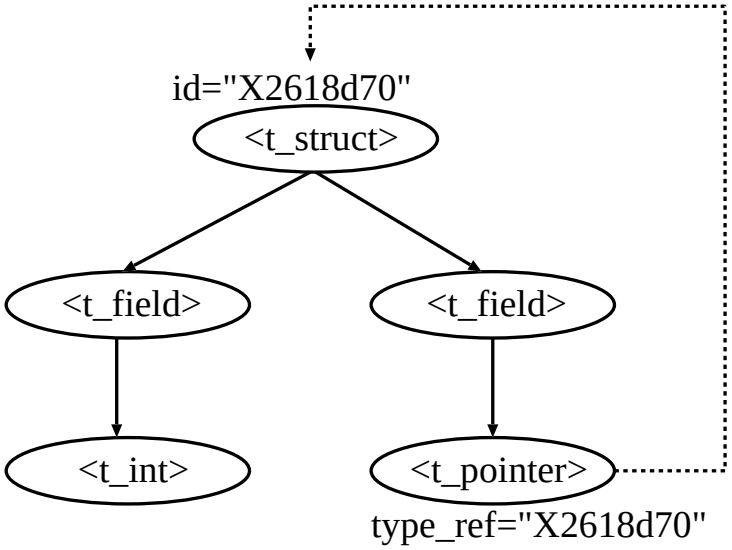

図 4.3: リスト構造を表す有向グラフのサイクル

このように、型に関する情報は、同じ意味を持つ異なったコードにも、各々ユニークに XML で表現し、その上、型の構造は別の木構造によって、表現し型の同値性を判定することができる。また、再帰的な型宣言も容易に表現する。

ANSI C のシンボル情報　シンボルのための情報は、単純に子供を持たない<symbol>エレメントを並べることで表現する。シンボルは、大別すると外部シンボル (external symbol) とローカルシンボル (local symbol) の 2 種類あり、前者は<translation␣union>エレメントの直下で、後者は<statement rhs="compound">エレメントの直下でリニアに表現する。例えば、外部宣言'int x' に対するシンボル'x' は、次の ACML 文書[10]で表現する。

[10]<translation␣union>エレメントの属性は省略する。

```
<translation_unit>
  <symbol name="x" type_ref="X80a57b8" ast_ref="X8084da8"
          namespace="normal" namelevel="file" is_tentative="true"
          is_incomplete="false" is_declaration="false"
          is_definition="false" is_global="true" is_enum_const="false"
          is_typedef="false" is_static="false" is_argument="false"
          is_builtin="false" />
</>
```

<symbol>エレメントには、13 種類の属性がある。<symbol>エレメントの属性は、名前の綴りや名前の種別 (変数、関数、型など)、スコープに関する情報などを直接的、あるいは間接的に提供する。例えば、ANSI C は互いに競合しない 4 つのネームスペースを持つことから、'namespace' 属性で、その 4 つの内のいづれか (上の'x' は、'normal') を示し、'namelevel' 属性で、関数の外側で宣言されたか否か (上の'x' は、'file') を示す。また、'ast_ref' 属性と'type_ref' 属性は、各々、構文情報部と型情報部への参照を与える。

このように、<symbol>エレメントの属性は、一般の手続き型言語を処理する際に用いる記号表の役割をする。<symbol>エレメントは、階層構造を持たないが、それ自身で多くの情報を提供することができる。

ACML における ID/IDREF の役割　上述した「ANSI C の構文情報」と「ANSI C の型情報」の部分で、我々は ACML において、ID/IDREF 属性を適宜使ってきた。ID 型の属性は、エレメントを一意に識別するためのものである。これに対応して、IDREF 型の属性を使えば、ID 値による参照ができるのである。一口に ID/IDREF 属性と言っても、ACML においては、その用途は大きく分けて 4 通りある。それは、シンボル情報部から構文情報部、型情報部へのリンクと制御フローのためのリンク、再帰的な型宣言のためのリンク、参照から定義へのリンクである。これらは、木構造だけでは、表現が難しい情報 (例えば、変数の定義・参照関係) を簡単に表現できる。以下で、この 4 つの機能の詳細を述べる。

- **シンボル情報部から構文情報部、型情報部へのリンク**
  あるシンボルについて、それが抽象構文木のどの部分を形成し、どんな型の構造をしているか確認するためのリンクである。実際には ACML で、<symbol>エレメントの'type_ref' 属性は、対応する<type>エレメントを参照し、'ast_ref' 属性は、抽象構文木における<declarator>エレメントを参照する。例えば、上述の'int x;' の'x' については、次の ACML 文書[11]で示す ID/IDREF 属性を記述する。

[11] <symbol>エレメントの'name' 属性と'type_ref' 属性、'ast_ref' 属性、一部の<declaration>エレメントの'id' 属性、type エレメントの'id' 属性以外の属性は省略する。

```
<!-- シンボル情報部 -->
<symbol name="x" type_ref="X80a57b8" ast_ref="X8084da8"/>
                                                    <!-- IDREF -->

<!-- 構文情報部 -->
<declaration>
  <int/>
  <declarator id="X8084da8">                        <!-- ID -->
    <declarator>
      <identifier> x </></></></></>

<!-- 型情報部 -->
<type id="X80a57b8">                                <!-- ID -->
  <t_prim>
    <t_int/></></>
```

- **制御フローのためのリンク**
  プログラム中の条件分岐部以外のところで、明示的に制御フローを変えることがある。例えば、while 文の中に、break 文を導入して、ループを脱出する場合である。一般に、ジャンプ文 (goto 文、break 文、continue 文、return 文) の種類や出現場所により、制御の変化、つまり、ジャンプ先が異なる。このため、ジャンプ文が出現する度に、その制御フローの変化を ID/IDREF 属性を使って簡単に表現[12]する。例として、次の goto 文を含むプログラムの断片を挙げる。

```
for (i = 0; i < 10; i++) {
  for (j = 0; j < 5; j++) {
    if (buf[i][j] < 0)
      goto EXIT_LOOP;
  }
}
EXIT_LOOP:
```

上の goto 文の行き先であるラベル付き文 (EXIT_LOOP) へのリンクは、次の ACML 文書[13]になる。

[12] return 文は、どこで出現しても制御を必ず呼び出し元に戻すから、特別 ID/IDREF 属性を用いない。
[13] <statement>エレメントの'rhs' 以外の属性は省略する。

```
  <statement rhs="for">
    <!-- for 文の最初にある 3 つの式は省略 -->
    <statement rhs="compound">
      <statement rhs="for">
        <!-- for 文の最初にある 3 つの式は省略 -->
        <statement rhs="compound">
          <statement rhs="if">
            <!-- if 文の条件式は省略 -->
<!-- goto 文 -->
            <statement rhs="goto" goto_ref="X80899c8">   <!-- ID -->
              <identifier>
                EXIT_LOOP  </></></></></></></></></>
<!-- ラベル付き文 -->
  <statement rhs="label" id="X80899c8">                <!-- IDREF -->
    <identifier>
      EXIT_LOOP  </></>
```

上の例では、goto 文の行き先 ('goto_ref') を、ID 値'X80899c8' で判別することによって、適切なラベル付き文への参照を与えている。また、switch-case 文についても、case 文と default 文がラベル付き文の一種であることから、上の例と類似したリンクを持つ。よって、制御フローを示す属性は次の 5 種類になる。

(1) goto_ref　(2) case_refs　(3) default_ref
(4) break_ref　(5) continue_ref

- **再帰的な型宣言のためのリンク**
  一般に、型解析を行なう際、型構成子 (type constructor) を用いて型を有向グラフで表現する。XML では、木構造以外に ID/IDREF 属性を使うことによって、グラフ構造を表現することもできる。上述した「ANSI C のための型情報」において、リスト構造を始めとする再帰的な型宣言を表現する場合も同様で、サイクルを含む有向グラフを用いた。複雑になったグラフも、<type>エレメントの'type_ref' 属性によって容易に表現できる (図 4.3)。

- **参照から定義へのリンク**
  プログラム間の要素関係、例えば、変数の参照と定義との対応を付けるために、ID/IDREF 属性を使ってリンクを張る。簡単な例を示すと、次のプログラムの断片は、

```
int x;          /* declaration */
return x;       /* statement */
```

次の ACML 文書[14]になる。

```
<declaration>
  <int/>
  <declarator id="X25f8100">           <!-- ID -->
    <declarator>
      <identifier ref="X25f8100">
        x </></></></></>
<statement rhs="return">
  <expression>
    <identifier ref="X25f8100">        <!-- IDREF -->
      x </></></>
```

<statement>エレメント以下にある<identifier>エレメントの'ref' 属性が示す値'X25f8100' は、'x' の参照部分からその定義部分へ対応付けを示している。より現実的な例として、上述した「ANSI C のための構文情報」で示した ACML 文書の ID/IDREF 属性がある。これにより、プログラム間の要素関係を容易に把握することができる。

4.2.3 XCI : Experimental C Interpreter

この節では、ANSI C プログラムから ACML 文書へのトランスレータ XCI の概要と、その開発過程で考案した ACML の定義に至った要因について述べる。そして、アンパーサを作成し、この XCI と ACML の検査の行なったこと、現在までに確認している XCI で実現していない機能を示す。

XCI の概要

XCI[5] は、我々が開発した ANSI C プログラムから ACML 文書へのトランスレータである。XCI は、'--xml' オプションを付けて起動すると ANSI C プログラムを ACML 文書に変換する。この変換は、上述した 4.2.2 節において、幾つかの例で示した通りである。

この他に XCI は、ANSI C インタプリタとしての機能も兼ね備える。トランスレータとは異なり、起動時に'--xml' オプションを外すことで、XCI は ANSI C インタプリタと

[14]<declarator>エレメントの'id' 属性と<identifier>エレメントの'ref' 属性、<statement>エレメントの'rhs' 属性以外の属性は省略する。

して動作する。対象プログラムの main 関数が仮引数を必要とする場合は、'--args' オプションを付けて起動する。例として、次の関数 (arg.c) を挙げる。

```
int main(int argc, char *argv[])
{
  int i;
  for (i = 0; i < argc; i++)
    printf("argv[%d] = \"%s\"\n", i, argv[i]);
  return (0);
}
```

このプログラムは、仮引数 argv の内容を単にコマンドラインで表示するものである。これを実行すると、次のようになる。

```
% xci.exe arg.c --args 10 20 30
argv[0] = "arg.c"
argv[1] = "10"
argv[2] = "20"
argv[3] = "30"
%
```

XCI は、約 14,000 行の ANSI C プログラムから成り、現在、Windows2000 上の Cygwin1.3.6 と Solaris8 の下で動作する。XCI の最初のバージョンを実現するのに、約 3ヶ月を費した。

ACML 導入の要因となった XCI

4.2.2 節で示した ACML は、この XCI の開発の中で考案したものである。特に、そのきっかけとなったのが、XCI の内部データの設計であった。開発の当初、プログラマが内部の抽象構文木データ構造に直接アクセスできる API を持つように XCI を設計した。しかし、我々の意見では次の 2 つの理由から、そのような API には、CASE ツールプラットフォームを構築する上で不利な点があると考えた。

1. プログラマが習得するのに多大なコストがかかる。
2. XCI の内部実装に依存しやすい。

つまり、仮に CASE ツール自体を作成のためのコストが削減できるとしても、新たに、XCI の内部データに対する API を習得するためのコストが発生してしまうことになる。ま

た、XCI の内部実装に依存した API は、XCI の変更や修正の度に API も変更や修正をしなければならなくなる。

そこで、3.2 節でも述べたように、我々は、データ統合に注目している。データ統合にを考慮すると、API を通じて内部データにアクセスするよりも、内部データのための共通データフォーマットを提供する方が望ましい。よって、ACML を導入し、'`--xml`' オプションを付けて起動するとき、ACML 文書を出力するように XCI の設計を変更した。

アンパーサ作成による ACML と XCI の検査

我々は、ANSI C のための CASE ツールプラットフォームを使った最初の事例として、アンパーサを作成した。アンパーサは、XCI とは反対に ACML 文書を元の ANSI C プログラムに変換する。この機能は、CASE ツールプラットフォームの基盤技術の 1 つになり有用である。アンパーサ作成の目的は、主に次の 3 点であり、第 5 章の CASE ツール作成実験を行なう上でも、確認しなければならない重要な事である。

1. ACML は、忠実に ANSI C 構文規則を表現しているか確認する。
2. XCI が DTD に従った ACML 文書を生成しているか確認する。
3. 既存の XML 関連ツールを使って、ACML 文書を処理できるか確認する。

アンパーサの実現は、XCI によってタグ付けた ACML 文書が十分な情報を持ち合わせているか否かを示す基準の１つとなった。実際 ACML は、制御フローを示す ID/IDREF 属性に誤ったリンクを発見したので、一部の修正を加えた。これは、当初、誤って `goto_ref` 属性のデフォルト値を `REQUIRED`、つまり、必ず属性値を与えなければならないよう設計していた。この場合、任意の `statement` エレメントの属性には、不要な `goto_ref` 属性が必ず設定されてしまい、プログラムの要素関係に意味のないリンクが多く存在し、正確な解析の妨げとなっていた。そこで、`goto_ref` 属性のデフォルト値を `IMPLIED` に修正し、必要に応じて、`statement` エレメントの属性に、`goto_ref` 属性を与えることができるようにした。XCI は、Solaris8 の下で動作できように修正を加えた。これは、XCI の開発を Windows2000 上の Cygwin1.3.6 で行なったため、多少の移植作業が必要になったのである。これは、ヘッダファイルと前処理系の独自拡張に起因する作業であった。XCI は、前処理後の ANSI C プログラムに対して、タグ付けを行なうため、前処理系は、既存のものを使い、その後のフェーズの処理系に関しては、自前で用意した。このため、本来、コンパイラが想定していた関数、あるいは変数 (例えば、__attribute__(arg) や __builtin_next_arg() など) に適宜処理を施した。また、XML 関連ツールの試用については、ACML の抽象構文木の部分から、DOM による操作でタグを取り除き、予約語や括弧を必要に応じて適切に書き加えた。結果として、アンパーサ作成では、DOM は API として十分な働きをした。

これにより、ACML 文書は ANSI C の構文に関する情報を十分に持ち合わせ、処理を施すこともできたことから、CASE ツール作成実験を行なえることを確認した。

現在の XCI で対応してない機能

上のアンパーサの実現により、ANSI C 構文規則については、十分な情報を持ち合わせていることを確認した。しかし、現在の XCI では、サポートしていない機能がある。それは、主に次の 3 つである。

- 前処理命令 (例えば、`#define`)
- ライブラリ関数 (例えば、`signal`)
- システムコール (例えば、`fork`)

前処理命令に関して、XCI は前処理後の ANSI C プログラムに対してタグ付けをする。そのため、前処理系が持つ機能 (例えば、条件付きコンパイルやマクロ定義) は、ACML において未定義の状態である。例として、次のソースプログラムを挙げる。

```
#include <stdio.h>
#define SIZE 10

int main()
{
  int a[SIZE], i;                                    /* SIZE */
  for(i = 0; i < SIZE; i++){                         /* SIZE */
    printf("a[%d] --> %u\n", i, &a[i]);
  }
  return 0;
}
```

これを XCI によって、ACML 文書に変換し、これをアンパースすると次のプログラム出力する。

```
#include <stdio.h>
int main()
{
int a[10], i;                                      /* SIZE --> 10 */
for (i = 0; i < 10; i++ ) {                        /* SIZE --> 10 */
printf("a[%d] --> %u\n", i, &a[i]);
}
return 0;
```

```
    }
```

ソースプログラムで定義した'SIZE'は、アンパースすると展開されたまま、'10'としてプログラム中に現れる。この両者は、表示は異なるが実行結果は等価である。また、字句情報 (例えば、コメント) も復元できない。

C のライブラリ関数とシステムコールに関しては、各々の意味を表現することが非常に難しい。例えば、C 標準ライブラリ関数である signal は、割り込み、つまり、シグナルが検出されたとき、どのように処理するかを決定する。signal は、2 つ引数 (対象シグナル名とそのシグナルハンドラへのポインタ) を持ち、その対象シグナルが起こる時、シグナルハンドラは呼び出される。このため、signal はプログラムの制御フローに影響を及ぼす可能性がある。その上、シグナルはプログラム中の至るところで起こり得る。つまり、signal の扱いには、難しい点が幾つか存在するのである。例えば、signal 呼び出しを含むプログラムをスライシングする場合、次の 2 つの困難が生じる。

- どのようにして、制御グラフを構成するか。
- ACML 文書として、この signal の意味をどのようにして、表現するか。

この種の議論は、'fork'、あるいは'execve を始めとするシステムコールにおいても同様である。

第5章　CASEツール作成実験

XML を用いた CASE ツールプラットフォームを基に、実際に CASE ツールを作成した。本研究では、CASE ツールにおいて基本的、かつ必要性の高い静的解析ツールであるプログラムスライシングツールとクロスリファレンサを実験的に実現した。どちらもプログラム要素間の関係を処理するため、CASE ツールプラットフォームの有効性を計る例題として適切である。この実装実験では、各々の実現は開発者 1 人で、わずか約 2 週間ですみ、開発コストの削減を確認した。

5.1　スライシングツール

5.1.1　プログラムスライシングの概要

プログラムスライシング [9] は 1982 年にメリーランド大学の Mark Weiser によってはじめて提案された部分プログラムを抽出する技術である。当初はデバッグの支援が目的であったが現在ではプログラム理解やテストケースの抽出、メトリックスなど、広い範囲に応用が試みられている。

スライシングとはスライスを求める技術のことである。スライスとはプログラム中のある命令のある変数に注目し、その変数に関しては元のプログラムと同じ値を計算する部分プログラムのことを言う。この注目する変数のことをスライシング基準という。スライシング基準においてもとのプログラムと同じ値を計算するプログラムであればスライスなので、一つのプログラムの、一つのスライシング基準に対してもスライスは、複数存在することになる。また、この考え方からもとのプログラムもスライスの一つである。しかし、スライシングはプログラムの要点を抽出する技術とも言えるのでスライスは可能なかぎり小さい方が望ましい。小さいスライスの方がより正確なスライスということができる。どのようにしてスライスを正確に求めるかはスライシングにおける重要な問題の一つである。

スライスはもとのプログラムからスライシング基準の値を計算するのに必要ない命令を 0 個以上削除することによって得られる。不要な命令はスライシング基準と間接的にもデータ依存関係や制御依存関係がない命令である。したがって、一般的にスライスはスライシング基準からデータ依存関係と制御依存関係をたどることによって得ることができる。

また、依存関係を前向きにたどるか後ろ向きにたどるかでスライシングは意味すると

ころが変化する。後ろ向きにたどることを後ろ向きスライシングといい、スライシングといったばあいはこちらを指すことが多い。スライシング基準となる変数の値に影響を与えた命令を抽出することになり、これはデバッグなどに応用されている。一方、前向きにたどることを前向きスライシングという。これはスライシング基準が影響を与える命令を抽出することになり、インパクトアナリシスなどに応用されている。

他にも、スライシングは2種類に分類できる。静的スライシングと動的スライシングである。この2つの違いは、入力を限定するかしないかである。限定しない方を静的スライシングといい、限定する方を動的スライシングという。入力を限定しない場合プログラムを静的に解析しなければならず、逆に入力が限定されていればプログラムを動かすことが可能なので動的に解析できるからである。以下に各々の特徴を簡単に示す。

- 静的スライシング
  - 任意の入力に対して元のプログラムと同じく動作する
  - 探索範囲が広い
  - 正確なスライスを求めるのは困難
  - 実現は動的スライシングツールに比べて容易
- ダイナミックスライシング
  - 特定の入力でしか元のプログラムと同じく動作しない
  - 探索範囲を著しく限定されている
  - 正確なスライスを求めやすい
  - 実現は静的スライシングに比べて困難

このように、プログラムスライシングとは、プログラムの中のある変数に影響を与える全ての文を抽出できるため、プログラムのデバックやプログラムの理解支援に非常に役に立つ有用な技術である。現在でも活発な研究が行われている。

しかし、多くは小さな言語が対象で、実際のプログラム言語を対象にした研究は少ない。フルセットに近いもの、例えば、Wisconsin Program Slicing Tool[31]、Unravel[32] は、XML を用いていない。本研究では、ANSI C 言語のフルセットに対応するスライシングを目指している。

5.1.2 スライシングツールの実現

現在、制限を加えた ANSI C 言語を対象に、Weiser のプログラムスライシングを行う CASE ツールを実現した。ここでは、本研究で用いた静的スライシングについて概要を示し、それに基づいて、実現したスライシングツールについて述べる。

静的スライシング

定義　ある変数に関しては、元のプログラムと同じ値を計算する実行可能な部分プログラムを静的スライスという。静的スライスは、プログラムのフローグラフをデータ依存関係と制御依存関係に従い、それを辿ることで得ることができる。プログラムの制御の流れる方向を示した有効グラフがフローグラフである。一般に、フローグラフは以下のように定義される。

1. **ノード**
   ノードはプログラム内の命令を表す
2. **アーク**
   ノード s と t の間のアークは、命令 s を実行したあと、制御が直接ノード t に移る可能性があることを示している。

例えば、階乗を計算するプログラムのフローグラフを図 5.1 の示す。

```
1    sum=0;
2    fact=1;
3    i=1;
4    while(i<=10){
5      sum=sum+i;
6      fact=fact*i
7      i++;
     }
8    printf("%d, %d\n", sum, fact);
```

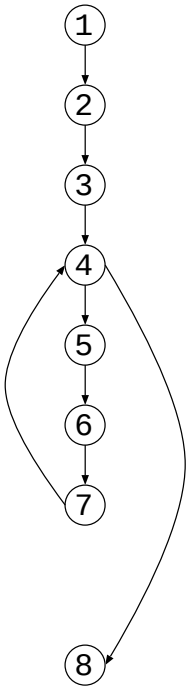

図 5.1: フローグラフの例

また、静的スライシングにおけるデータ依存関係と制御依存関係の定義を次に示す。

- **データ依存関係，**$DD(s,t)$
  変数 w が存在するとする。命令 s において定義された変数 w が、w を参照している

別の命令 t に到達するとき、「s から t に対してデータ依存関係 $DD(s,t)$ がある」という。ここで言う「到達する」とは、命令 s が変数 w を定義し、かつ、フローグラフ上で s から t に至るパスが存在し、そのパス上では変数 w が再定義されないことを言う。つまり、データ依存関係があることは、s で定義した変数を t で参照する可能性があることを意味している。

- **制御依存関係，** $CD(s,t)$
  命令 s が分岐命令またはループ命令で、命令 t がその文の内部に直接含まれているとき、「s から t に対して制御依存関係 $CD(s,t)$ がある」という。ここで言う「直接含まれている」とは、ネストした分岐文やループ文に中に t が含まれていないことを意味する。つまり、制御依存関係があることは、命令 t が実行されるかどうかが命令 s の実行結果に依存していることを意味する。

方法　静的スライシングではスライシング基準はプログラム中の命令 u とプログラム中の変数の部分集合 V によって与えられる。これを $C = (u, V)$ とする。スライシング基準 $C = (u, V)$ に関する静的スライス SS の求め方は、次のように定式化できる。

1. $A^0 =$ { 命令 t| ある変数 $v \in$ V に対して命令 t における変数 v の定義が命令 u に到達する、あるいは $CD(t,u)$}
2. $i \geq 1$ に対して、$A^1 =$ { 命令 s| ある命令 $t \in A^{i-1}$が存在して $DD(s,t)$ もしくは $CD(s,t)$}
3. $SS = \bigcup_{i=0}^{\infty} A^i \bigcup \{u\}$

実現したスライシングツール

XML を用いた CASE ツールプラットフォームを使用することで、プログラム中の条件を満たす変数の抽出と必要な文の抽出は比較的容易だった。DOM[10] を用いたことも開発を容易にした。このツールは約 2,000 行の Java プログラムで、作成期間は、開発者 1 人で約 2 週間を費やした。その仕様を以下に示す。

- スライシングの基準として、任意の文の任意の 1 変数が指定できる。
- if-else 文、および while 文を持つプログラムから実行可能な必要部分を抽出する。switch-case 文、goto 文、for 文は扱わない。
- ポインタ、配列の解析、および関数間を跨る変数の解析は行わない。

例えば、次のファイルの行数と語数、文字数を計算する関数を対象にスライシングを行ったところ、行数部、語数部、文字数部のみを取り出せた。

```
 1  iw = 0; /*in a word*/
 2  nl = 0;
 3  nw = 0;
 4  nc = 0;
 5  c = fgetc(stdin);
 6  while (c != EOF) {
 7    nc = nc + 1;
 8    if (c == '\n')
 9      nl = nl + 1;
10    if (isspace(c))
11      inword = 0;
12    else if (iw == 0) {
13      inword = 1;
14      nw = nw + 1;
15    }
16    c = fgetc(stdin);
17  }
18  printf("%d\n", nl);
19  printf("%d\n", nw);
20  printf("%d\n", nc);
```

元のプログラム

```
 2  nl = 0;

 5  c = fgetc(stdin);
 6  while (c != EOF) {

 8    if (c == '\n')
 9      nl = nl + 1;

16    c = fgetc(stdin);
17  }
18  printf("%d\n", nl);
```

スライス結果 (18, nl)

結果として、粗い比較にはなるが、XCI の構文解析、静的意味解析に費した労力 (2 人月) に比べると、大幅に少ない労力 (0.5 人月) で済んだので、この種の小さな CASE ツールの作成においては、開発コストが削減でき XML を用いる有用性を確認した。

5.2 クロスリファレンサ

5.2.1 クロスリファレンサの概要

クロスリファレンサとは、ソースプログラムの構成要素間にリンクを張り、その関係を明らかにするためのツールである。例えば、ソースプログラムの中から、指定した関数の定義部分と参照部分を見つけ出したり、複数のサブディレクトリや、プログラムから構成される大規模ソフトウェアを扱い、ファイル名からそのソースファイルへの参照を可能にする。例えば、既存のツールには、GLOBAL[33] や cxref[34] がある。

5.2.2 クロスリファレンサの実現

ACML と XCI を用いて、簡単な参照機能を持つ ANSI C プログラム用のクロスリファレンサを実現した。これは、フルセットの ANSI C 言語に対応している。表示は、既存の Web ブラウザを使用することにした。そのため、ソースプログラムを読むために HTML への変換が必要であった。ACML から HTML への変換は XSLT[11] で行った。これは、DOM でも可能だが、XSLT の利便性を探るために、今回は XSLT を活用した。スタイルシートは、約 2,000 行で、作成期間は開発者 1 人で約 2 週間を要した。その主な仕様を以下に示す。

- ソースプログラム中の関数呼び出し部分、変数や型の出現部分から、各々の定義部分への参照
- ファイル名のリストアップと、そのソースファイルへの参照
- 関数名、大域変数名、ローカル変数名、構造体と enum のタグ名、メンバ名のリストアップと、各々の定義部分への参照
- 自動整形機能 (ACML には、表現情報がないため)

実際に、ANSI C プログラム ([33] のソースコード) を表示したが、色分け、リンク先が正しく処理され、ACML から HTML への変換が成功したことを確認した。図 5.2 は HTML への変換結果の表示例である。

結果として、スライシングツールと同様、この種の小さな CASE ツールの作成においては、開発コストが十分に削減でき、XML を用いる有用性を確認した。

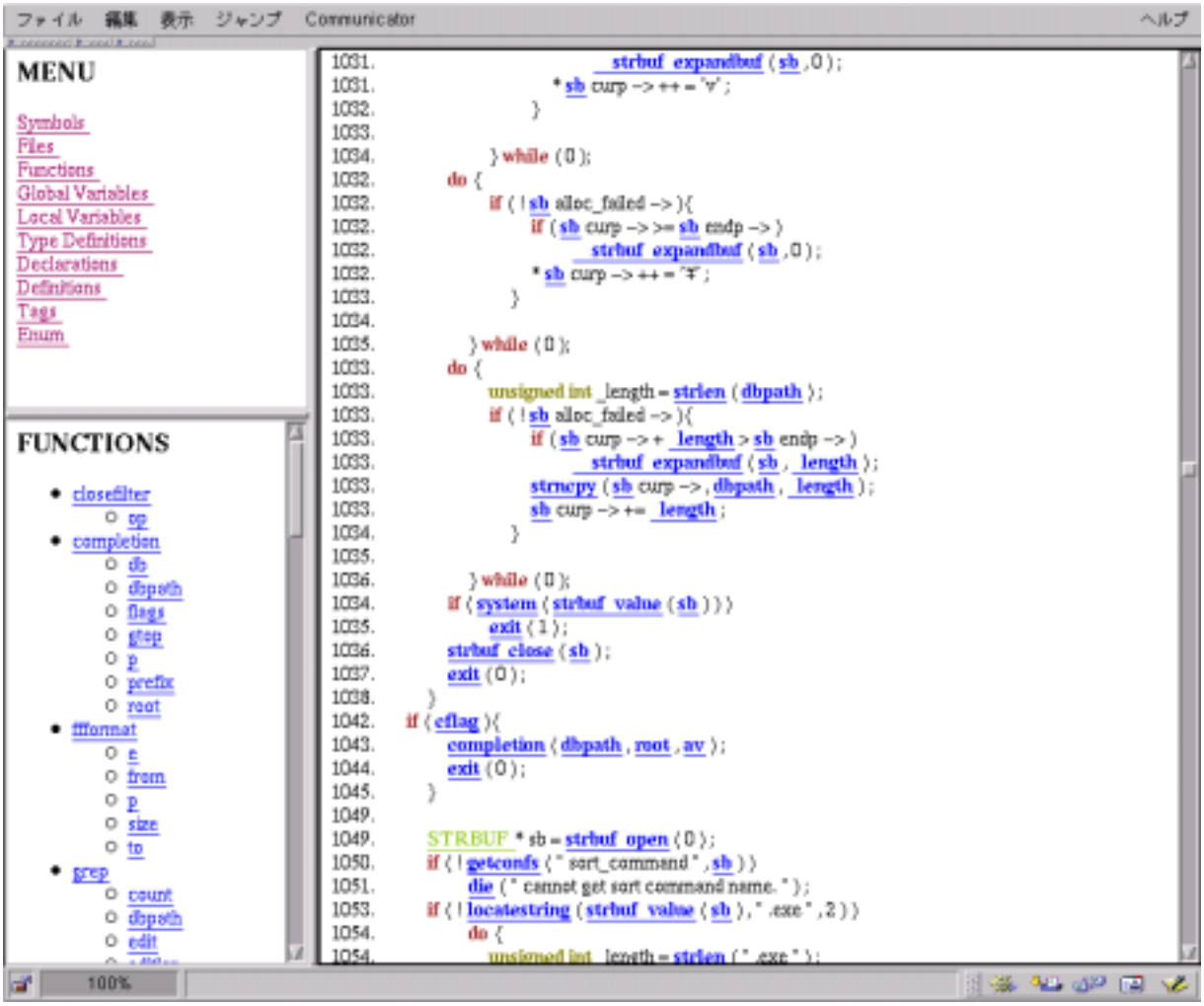

図 5.2: Web ブラウザによる表示例

第6章　議論

6.1　開発コスト削減の要因

実際のツール作成は、ACML と XCI、既存の XML 関連ツールの連携の下で行った。実際に、スライシングツールとクロスリファレンサの作成に費した時間は、各々1 人で 2 週間ずつで非常に短期間ですんだ。表 6.1 は、各々の開発コストをまとめたものである。

	作成期間	使用した XML 技術	コード数
プログラムスライシングツール	1 人で 2 週間	DOM	2000 行
クロスリファレンサ	1 人で 2 週間	XSLT	2000 行

表 6.1: 作成した CASE ツールの開発コスト

これらの CASE ツール作成から、我々は、次の 4 点が開発コストの削減に貢献したと考える。

- XCI で対象ソフトウェアの構文解析と静的な意味解析まで完了しているため、目的のツールの本質的な機能の実現に専念できた。
- データのやり取りが、XML である ACML で行われたため、XCI から特別なデータ変換をせず、容易に DOM や XSLT と連携がとれた。
- DOM や XSLT は、公開された汎用的な技術であるため、技術移転コストが低かった。
- ACML がデータ交換の簡潔な仕様として機能したので、プログラマ間のやり取りを少なくできた。実際に XCI の開発者と CASE ツール開発者の間のやり取りは、ACML を定義した DTD のみで、ほとんどすんだ。

6.2　ACML 文書の解読

一般に、XML 文書を扱うには、たいていの場合、対象とするファイルのサイズが大きく複雑なため、解読作業は困難である。XML を用いたアプリケーション開発においては、

仮にDTDによるスキーマの記述があったとしても、XML文書を直接、解読する作業が必要になる。この場合、最も手軽で多く用いるのがgrepツールである。XML文書には、プレーンテキストとしての特徴があるため、従来までのデータ形式では解読が困難であったCASEツールの内部データ(本研究におけるACML文書)を、この種のテキスト処理ツールを使って、より簡単に解読できるようになった。実際にCASEツール作成において、改めてテキスト処理ツールが使えるという利点を確認できた。

しかし、比較的規模の大きなXML文書の構造を把握したい場合、テキスト処理ツールだけでは、困難がある。確かにDTDによる表現は、コンパクトで非常に簡潔であるが、実際にXML文書を目にすると、慣れるまではその構造を把握するために、ある程度の労力が必要である。そこで、DomEcho[41]を始めとする木構造ビューアが、XML文書の構造を把握するのに役立つ。DomEchoは、約370行の小規模なJavaプログラムで記述した木構造ビューアとして、公開されている。図6.1は、実際にACML文書を表示させた例である。これは、木構造を直観的に表現し、ACML文書からANSI C構文構造を把握することを容易にした。実際にCASEツール作成の当初は、このDomEchoを参照しながら、DOMプログラミングを行なった。これにより、操作対象が視覚的に確認できるため、簡単なエラーや、プログラミングの労力を減らすことができた。

このように、ACML文書はプレーンテキストなので、プログラマにとっては、バイナリーフォーマットよりも、非常に分かり易い。既存のテキスト処理ツールを活用することもできるのことが利点である。しかし、実際にACML文書を前にすると、ファイルサイズがソースファイルよりも大きくなっているため、慣れるまでは、その文書構造を把握することが困難である。この問題を解決するために、DomEchoを用いることで、上の表示例(6.1図)のように、直観的にプログラム構造を把握することができる。よって、テキスト処理ツールや木構造ビューアを使うことによって、ソースファイルよりも大きく複雑になったACML文書を解読する労力を軽減するため、この解読作業がCASEツール開発コストに及ぼす影響を最小限に抑えることができる。

6.3 ACMLの改良の余地

ACMLは構文規則に忠実に表現してるため、プログラミングスライスとクロスリファレンスに必要な情報も一通り備わっていた。そのため、比較的容易に各々のツールを作成することができた。また、このツール作成の経験から得られた成果として、現時点のACMLの改良の余地を発見することができた。以下で、その主なものを示す。

6.3.1 エレメントノードの文脈のための情報とリンク

ACMLが表現する木構造には、幾つかの改良点があった。これらは、設計不足というよりも、実装を通じて初めて分かることである。DTDの設計には、多くのトレードオフがあるので、実装実験は重要である。

図 6.1: ACML 文書の表示例

- エレメントノードの文脈
  現在の ACML には、エレメントの文脈、特に、そのエレメントが何番目の兄弟であるかの情報が不足している。特に下方から上方へ解析を行う場合は、親に戻って子供を数えなければ知ることができない。例えば、図 6.2 で、ある `statement` が then 部か、else 部かは、その `statement`(B,C) の属性には書かれていない。木構造が大きくなれば、数えるのは面倒な定型作業である。これは、データ統合として ACML に含めるべきか、アプリケーションで対処すべきかなど検討中である。

- リンク
  ACML のリンク (ID/IDREF) による情報については、シンボル情報部から構文情報部と型情報部へは、十分に関連付けができている。しかし、現在の ACML では、そ

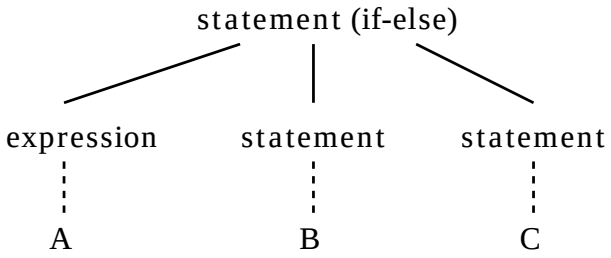

図 6.2: 3.2.1 節の例を木構造で表現

の逆向きのリンクがない。これも、ACML に含めるべきか、アプリケーションで対処すべきかは検討中である。

6.3.2 前処理に関する情報

前処理系は、ANSI C とは独立した文法 (例えば、マクロ定義や条件付コンパイル)[52] を持つ。XCI は前処理後の ANSI C プログラムにタグ付けするため、現段階ではマクロは展開されたまま復元不可能である。これでは、ソースプログラムに対して完全には操作できないし、解析結果もソースプログラムに完全には反映されない。マクロ定義を復元する手法を検討する必要がある。

これは、我々だけが抱える問題ではなく、ANSI C ツールにとって一般的な問題である。Sapid では、部分的 (`#include` 命令と`#define` 命令) に各ファイルに前処理前後間の対応表を生成することで解決している。

6.3.3 字句情報 (空白、コメント、インデント) の保存

ACML 文書をアンパースすると、元のプログラムと意味的には等価であるが、見かけは等価ではない。現在の ACML では、字句情報を保存してないからである。プログラミングスタイル [55] は、ソフトウェアを理解する上で貴重な情報となり、プログラマの意図も明確に表現する。字句情報の保存は、技術的には可能である。しかし、現在、その保存方法について、ACML のデータとして保存するか、あるいは、ACML とは独立して保存するかなど、その実現方法を検討中である。

6.3.4 DTD の決定性問題

現在の ACML を定義する DTD は、XML1.0[1] の仕様には合致しているが、推奨事項に反するエレメント宣言をしている。XML1.0 では SGML との互換性を考慮して決定的内容モデルを推奨しているが、ACML 用の DTD では決定的内容モデルになっていない。そ

のため、ACML 用の DTD では、正確な妥当性の検証ができない場合がある。しかし、制限のない拡張 BNF による DTD 表現は、簡潔で直感的に理解できる。仮に決定的内容モデルだけで、ACML を定義しようとすると、エレメントの数を大幅に増やさなければならず、DTD を複雑にするだけである。それゆえ、現在の ACML の表現の方に利点がある。

また、他のスキーマ言語 (例えば、RELAX[17]) は、非決定的内容モデルの記述を認めているものもある。これについては、十分に検討する必要があるが、次の点においては、DTD の方に利点があることから、我々は、DTD に対して積極的である。

- SGML で使われていた経緯から、成熟した技術で長所と短所が明確である。
- 安定ツールが、他のスキーマ言語よりも DTD の方を多くサポートしている。
- 文書規則をコンパクトに表現することができる。
- データ型はないが、エレメントの入れ子構造を簡潔に表現できる。

6.4 ACML に対する操作・編集

6.4.1 DOM の利便性

DOM[10] は、木構造を基に直感的に操作・編集することができるので、習得が早く使いやすかった。しかし、DOM には低レベルな操作・編集しか定義されてない。これらの操作・編集をそのまま用いるだけでは明らかに力不足であり、複雑な操作をするには、既存のライブラリを組み合わせなければならない。例えば、任意のエレメント名と属性値を基に、特定のエレメントを検索する複雑な操作は、エレメント名を参照する関数、属性値を参照する関数、木構造を適切に走査する関数などを組み合わせて定義しなければならない。このため、DOM による操作では、冗長な作業が生じることが多い。

6.4.2 XSLT の利便性

XSLT[11] の基本的な処理はパターンマッチングで、XPath[12] を使って XML 文書内の検索を容易に行える。例えば、6.4.1 節の例 (エレメント名と属性値から特定のエレメントを検索する) は、“`//elementName[@attribute='name']`” と記述するだけですむ。XSLT は、関数型プログラミング [36] の概念を基にしているため、経験があれば、非常に簡単に扱うことができる。しかし、スタイルシート自体、XML 文書なのでタグ付け作業が必要で、これはあまりに冗長的である。また、XSLT は関数型プログラミング特有の高階関数が使えないし、リストのようなデータ構造もない。このため、XSLT の能力だけでは、プログラムスライシングのような複雑な依存解析をするのは困難がある。

6.4.3　ライブラリの必要性

ツール作成実験を行った結果、ツール開発の際には、非終端ノードからノード検索や属性値の参照などの定型作業が生じる。ある程度複雑な作業には、その定型作業を含むライブラリがあると便利である。また、DOM と XSLT は、各々、一長一短であるため、これらの協調作業による処理は望ましいことである。この協調作業を促進するようなライブラリも必要である。これを踏まえ、XML を用いた CASE ツール作成に適したライブラリを開発中である。

第7章　関連研究

7.1　Sapid

Sapid[1][45] は名古屋大学の阿草清滋、山本晋一郎らによって提案、開発された細粒度リポジトリに基づいた ANSI C のための CASE ツール・プラットホームである。与えられた ANSI C ソースプログラムに対して、Sapid は、構文構造と静的意味の情報を I-model と呼ばれるフォーマットとしてファイルに格納する。Sapid は、独自で開発したソフトウェアデータベースとアクセスルーチン、作成したツール群から構成され、制御統合を実現している。元々Sapid は、XML を用いてないが、現在、XML を基づいた機能を組み入れようとしている [46]。ここでは、Sapid の概要について述べる。

Sapid の概要

Sapid のもっとも大きな特徴として細粒度リポジトリであることが上げられる。リポジトリはソフトウェアのライフサイクルで行われる様々な問い合わせや操作に対応できる必要がある。そのためには可能な限り細かな情報まで保持しているモデルに基づいたリポジトリであることが望ましい。不要な情報を切り捨てることは容易であるが逆は不可能なので、粒度の荒いモデルでは将来発生する可能性のある様々な要求に対応できないからである。また、粒度の粗いものを細かい粒度にするのは不可能であるが、細かい粒度を粗い粒度にするのは容易である。しかし、細かな粒度を設定すると管理するオブジェクトが増加し、検索性の低下など弊害も生じる。このトレードオフを鑑みて Sapid ではモデルとして I-model を提案し採用している。このモデルは構文規則に基づいて作られており、解析結果を用いて C 言語の文を直接解釈実行できる仮想機械を作れるほど細かい粒度を持つ。一方で、ソフトウェアのライフサイクルにおいて重要ではないものを切り捨てて、十分細かい粒度を持ちながら簡潔で扱いやすいモデルになっている。さらに、適当なビューを設けることにより、粒度を調節できるように考えられている。

Sapid は大きく分けて SDB、API 群、ツール群の 3 つの部分から成る。

- **SDB**
  Sapid の基盤となるソフトウェアデータベースを Sapid では SDB と呼んでいる。SDB は C 言語のソースプログラムを解析した結果を格納するソフトウェアデータベース

[1] Sophisticated Application Programming Interface for software Development

である。ソフトウェアの構成要素にはソースプログラム以外にも仕様書やマニュアルが含まれるが、Sapid では今のところそれらには対応していない。Sapid では C 言語のソースプログラムを幾つかののモデルに基づいて解析し、その結果を SDB に格納している。モデルは基本的に実体関連モデルで、C 言語の構文を実体と関連として捉えている。モデルには Sapid の基盤をなしている I-model や、前処理に関する P-model、メモリ領域と値に関する Area-model などがある。

- **API 群**
  Sapid では構築されたソフトウエアデータベースにアクセスし、それを CASE ツールの開発に利用するための手段として API を用意している。その中で定義されている関数は、データベースのスキーマを取得するメタデータ取得関数、オブジェクトや関連の属性値取得関数、オブジェクト取得関数、関連取得関数などである。

  - **AR**
    最も基本的な API が AR(Access Routines) である。この API には I-model と P-model の SDB に関する API である。AR は、SDB にアクセスするための API を提供している。その中で定義されている関数はデータベースのスキーマを取得するメタデータ取得関数、オブジェクトや関連の属性値取得関数、オブジェクト取得関数、関連取得関数などである。
  - **SMU**
    SMU は AR で良く使われるイデオムをまとめたものである。検索条件を満たすオブジェクトを集める配列取得関数や、構文木を与えられた順序で辿っていくトラバース関数、オブジェクトが関数の引数になっているか調べる関数などがある。
  - **SpdUtil**
    SpdUtil も Sapid で利用される汎用の関数をまとめたものである。標準ライブラリ関数と組み合わせて AR を使うときの便宜を図った関数が多く含まれる。メッセージ出力関数、メモリ操作関数、文字列操作関数、ファイル操作関数などがある。
  - **SDA**
    SDA(Sapid Dependency Analyzer) は、プログラムの依存関係を解析するための、ビューとそれに関する関数から成る。ビューはプログラムの制御依存関係と、データ依存関係を静的に解析し有効グラフでモデル化する。関数には初期化関数とデータ取得関数があり、ある変数がどこで定義された可能性があるか、変数がどこで参照される可能性があるかなどの情報を得ることができる。

- **ツール群**
  Sapid には Sapid を用いて CASE ツールを開発するのに必要なツールも用意されている。SDB をブラウズする sat4 や、ソースプログラムの理解支援ツール SPIE、SDB

に基づいた簡易 C 言語インタプリタ sint、関数仕様管理ツール mkSpec5、などである。

7.2 JavaML

JavaML[2] は、Java 言語のためのマークアップ言語の１つで、Greg J. Badros によって開発された。JavaML は、Java 特有の構文から独立して Java プログラムをモデル化しようと試みている。この抽象化により種々のオブジェクト指向言語を一定のフォーマットとして記述できる可能性を持つ。ソフトウェアプロダクトによっては、JavaML のような粗い粒度のモデルが適当な場合もあるが、下流工程を支援するには力不足である。ここでは、JavaML の概要について述べる。

JavaML の概要

従来のプレーンテキストでソースプログラムを表現する方法は、プログラマにとって便利ではあるが、プログラムの構造を解明するためには構文解析が必要である。例えば、grep を始めとする多くの簡単な支援ツールは、ソースプログラムの字句構造しか扱わない。この問題の根本的な原因の 1 つは、ソースプログラムの標準的な構造化表記が存在しないことにある。CASE ツールにとって、分析や操作がしやすいプログラム構造を直接的に表す汎用形式が必要なのである。XML は現在のソースプログラム表記方法の中で、この汎用形式として最も期待できる。

JavaML の DTD は、妥当な JavaML 文書の種々の要素と、要素を組み合わせる方法を指定する。エレメント、及びその属性と、それらが表すプログラム言語の構成体とは、自然な形で対応している。ソースプログラムの構造は、JavaML 文書の中でエレメントの入れ子として反映される。この表記方法を使えば、XML 文書や SGML 文書の操作や照会を行う多種多様なツールを活用して、Java ソースプログラム分析のためのオープンな基盤を構築することができる。そのため、JavaML は、CASE ツール用の標準的な Java ソースプログラム表記として最適である。

JavaML は従来の表記方法とほとんど変わらない強固さを持ちながら、数多くの弱点を克服している。従来のような文字ベースでのプログラム表記と違い、JavaML は、XML の構文を使って入れ子になったエレメントとしてプログラム構造を直接的に表現する。さらに、XML の ID/IDREF 属性のリンク機能を使用して、プログラムグラフで追加のエッジを表す。XML はテキストベースの表記なので、従来型ソース表記の利点の多くの部分が XML にも当てはまる。XML の適用分野の 1 つである JavaML は構文解析しやすく、XML を扱う既存のツールはすべて JavaML 表記の Java ソースプログラムに適用することができる。JavaML ツールは既存の基盤を利用し、標準的な表記を活用して相互運用性を高めている。

JavaML プロトタイプは、Java 言語を含む、幾つかのオブジェクト指向プログラミング

言語の言語構成要素を、言語特有の構文とは無関係な形でモデル化しようとするものである。Java 以外の対象言語としては、まず Java によく似た Smalltalk がある。さらに、従来の Java ソースプログラムや Smalltalk file-out 形式に変換できるフォーマットも含めることができる。こうした目標を念頭に置いて、JavaML は各構成要素の原理に基づいて設計された後、くり返し精錬されて、実用性の点で、また生成されるマークアップ言語の読みやすさの点で改善された。JavaML では、メソッド、スーパークラス、メッセージ送信、リテラル数などの概念が文書内容の要素と属性によって直接的に表される。

JavaML 表記の長さは、従来のソースプログラムの約 3 倍になる。基本的に、XML に変換する場合、このように長くなることは避けられない。開発やプログラム編集などの通常の作業では、プログラマは従来のコンパクトな表記を採用するだろうが、JavaML は従来のソースプログラムを補う役割を果たす。開発者が JavaML を表示して直接読むことも可能だが、JavaML は特にツールでの使用に適している。

第8章　おわりに

8.1　まとめ

我々は、CASE ツール開発のコストが高くなる原因が、その CASE ツールの内部データを他の CASE ツールで使えないことにあると考え、そのような内部データを統一的に処理することを目指し、XML を用いた CASE ツールプラットフォームを提案した。本研究では、特に下流 CASE のデータ統合に注目し、ANSI C プログラムのみをターゲットにした CASE ツールプラットフォームを実現した。これは、次に示す 3 つの構成要素から成り立つ。

1. ANSI C 構文規則を忠実に表現した ACML。
2. ANSI C ソースプログラムを ACML に変換する XCI。
3. 既存の XML 関連技術を使った CASE ツール。
   (アンパーサ、プログラムスライシングツール、クロスリファレンサ)

ツール作成実験として、この CASE ツールプラットフォームを基にプログラムスライシングツールとクロスリファレンサを実験的に作成をした。これらの実験は、CASE ツールプラットフォームの有効性を計る例題として適切であった。その主な理由は、これら 2 つの CASE ツールが、次に示す特徴を持つからである。

- 他の CASE ツールに比べ、非常に基本的、かつ必要性の高い静的解析ツールである。
- 解析において、プログラム要素間の関係を処理する必要がある。

これらの実験において、各々の実現は開発者 1 人で、わずか約 2 週間ですんだ。これに対し、XCI の ANSI C の構文解析器と静的意味解析器の部分の実現には、開発者 1 人で、2 カ月を費やした。それゆえ、ACML と XCI を使ったことで、本来、必要であるはずの 2 カ月の開発期間を削減し、この種の小さな CASE ツールの作成においては、XML を用いる有用性を確認した。また、この実験で得られた経験として、設計した DTD の改良や、ツール作成の際に生じる定型作業のライブラリ化など、検討すべき点を発見できた。

8.2 結論

開発期間を約 2 カ月削減できた要因として、次に挙げる 4 点を確認した。

- XCI で対象ソフトウェアの構文解析と静的な意味解析まで完了しているため、目的のツールの本質的な機能の実現に専念できた。
- データのやり取りが、XML である ACML で行われたため、XCI から特別なデータ変換をせず、容易に DOM や XSLT と連携がとれた。
- DOM や XSLT は、公開された汎用的な技術であるため、技術移転コストが低かった。
- ACML がデータ交換の簡潔な仕様として機能したので、プログラマ間のやり取りを少なくできた。実際に XCI の開発者と CASE ツール開発者の間のやり取りは、ACML を定義した DTD のみで、ほとんどすんだ。

これにより、XML を CASE ツールプラットフォームに適用することが、有効であることが確認できる。CASE ツールプラットフォームとして、基本的で重要な機能である各 CASE ツールで共通して使うデータを統合し、提供することが実現できたと我々は考えている。

しかし、定量的な評価を行っていないこと、実験対象が小規模であったこと、実験事例が少ないことから、その有効性が十分に示せたわけではない。本研究における実験を通じても、我々は、上で述べたコスト削減の要因の他に検討すべき点を確認した。その主なものは、次の通りである。

1. ACML には改良の余地があることを確認した。例えば、あるノードに対して、前方には幾つのノードが存在するか示す属性がない。これは、木構造内の走査、特に上方へ辿るときに非常に便利である。
2. DOM や XSLT の利便性は高かったが、CASE ツール開発において、共通に使うことができるライブラリを構築する必要性を確認した。例として、次の操作を挙げる。
   - 特定の識別子を持つステートメントをすべて抽出する。
   - 特定の識別子を含んだ最も深い部分のステートメントを抽出する。

以上のことを踏まえ、本論文では、対象を ANSI C に制限したことや、実装実験の回数が少なく小規模ではあったが、XML は CASE ツール開発におけるデータ統合技術に有用であると考えることができる。今後も、さらなる実装実験を行い、検討すべき点を整理し、適宜改良を施すことで、この CASE ツールプラットフォームを洗練する必要がある。これにより、XML の有用性を十分に示し、本質的に困難であったデータ統合を実現させることは、ソフトウェア開発において重要なことである。

8.3 今後の課題

今後の主な課題を以下に示す。

- より制限の少ない静的スライシングツールの実現。
- 動的解析に対応する ACML の拡張。
- Java や C++など他の言語へ応用。
- ACML を使った他の CASE ツールの作成。(例えば、テストケース生成器)
- 前処理命令や字句情報 (例えば、コメント、コーディングスタイル) の保存、システムコール、インラインアセンブラへの対応。
- ACML 用の版管理システムの構築。
- DOM と XSLT を補うライブラリの構築。
- ACML と XMI の連携手法の構築。
- 未完成 (バグだらけ) のプログラムやプログラマの意図に対する適切な表現の定義。

謝辞

本研究を進めるに当たり、最後まで熱心に御指導を頂きました権藤克彦助教授に深く感謝致します。また、数々の助言、励ましを頂いた片山卓也教授を始め、ソフトウェア基礎講座の皆様に深く感謝すると共に、厚く御礼申し上げます。最後に、修士論文を共に戦い、励ましあった仲間たちに感謝します。

付 録A　ACML DTD

```
<!-- ACML: DTD for ANSI C program -->
<!-- entities for element -->
<!ENTITY % storage_class_specifier "(auto | register | static | extern | typedef)">
<!ENTITY % type_specifier          "(void | char | short | int  | long | float | double
                                   | signed | unsigned | struct_or_union_specifier
                                   | enum_specifier | typedef_name)">
<!ENTITY % type_qualifier          "(const | volatile)">
<!ENTITY % declaration_specifiers  "(%storage_class_specifier; | %type_specifier;
                                   | %type_qualifier;)+">
<!ENTITY % expression_opt          "(expression | empty_expression)">

<!-- entities for attributes -->
<!ENTITY % attr.common             "id            ID          #REQUIRED
                                    lineno        CDATA       #REQUIRED
                                    filename      CDATA       #REQUIRED">
<!ENTITY % attr.ref                "ref           IDREFS      #REQUIRED">
<!ENTITY % attr.rhs                "rhs           CDATA       #REQUIRED">
<!ENTITY % attr.exp                "type_ref      IDREF       #REQUIRED">
<!ENTITY % attr.control            "goto_ref      IDREF       #IMPLIED
                                    case_refs     IDREFS      #IMPLIED
                                    default_ref   IDREF       #IMPLIED
                                    continue_ref  IDREF       #IMPLIED
                                    break_ref     IDREF       #IMPLIED">

<!-- element definitions and attribute declarations for ANSI C syntax-->
<!ELEMENT translation_unit      (symbol*, (function_definition | declaration)+, type*)>
<!ATTLIST translation_unit      %attr.common;>
<!ELEMENT function_definition   ((%declaration_specifiers;)?,
                                   declarator, declaration*, statement)>
<!ATTLIST function_definition   %attr.common;>
<!ELEMENT declaration (%declaration_specifiers;, (declarator, initializer?)*)>
<!ATTLIST declaration           %attr.common;>
<!-- rhs = "struct" or "union" -->
<!ELEMENT struct_or_union_specifier   (identifier?, struct_declaration*)>
<!ATTLIST struct_or_union_specifier   %attr.common; %attr.rhs;>
<!ELEMENT struct_declaration    ((%type_specifier; | %type_qualifier;)+,
                                        (declarator?, expression?)+)>
```

```
<!ATTLIST struct_declaration    %attr.common;>
<!-- rhs = "notag", "normal" or "nobody" -->
<!ELEMENT enum_specifier        (identifier?, (identifier, expression?)*)>
<!ATTLIST enum_specifier        %attr.common; %attr.rhs;>
<!-- rhs = "pointer_null", "pointer", "id", "paren", "array", "func_new", "func_old" -->
<!ELEMENT declarator            ((pointer?, declarator) | identifier | declarator
                                | (declarator, expression?)
                                | (declarator, parameter_declaration+, ellipsis?)
                                | (declarator, identifier*))>
<!ATTLIST declarator            %attr.common; %attr.rhs;>
<!-- rhs = "single" or "pair" -->
<!ELEMENT pointer               ((%type_qualifier;)*, pointer?)>
<!ATTLIST pointer               %attr.common; %attr.rhs;>
<!-- rhs = "dec" or "abs" -->
<!ELEMENT parameter_declaration (%declaration_specifiers;,
                                 (declarator | abstract_declarator?))>
<!ATTLIST parameter_declaration %attr.common; %attr.rhs;>
<!-- rhs = "pointer_nil", "pointer_pair", "paren", "array_nil",
          "array_pair", "func_nil", "func_pair" -->
<!ELEMENT abstract_declarator  ((pointer?, abstract_declarator?) | abstract_declarator
                               | (abstract_declarator?, expression?)
                               | (abstract_declarator?, parameter_declaration*, ellipsis?))>
<!ATTLIST abstract_declarator  %attr.common; %attr.rhs;>
<!-- rhs = "exp" or "list" -->
<!ELEMENT initializer           (expression | initializer+)>
<!ATTLIST initializer           %attr.common; %attr.rhs;>
<!ELEMENT type_name             ((%type_specifier;
                                | %type_qualifier;)+, abstract_declarator?)>
<!ATTLIST type_name             %attr.common;>
<!-- rhs = "label", "default", "expression_opt", "return", "compound", "if", "if-else",
          "case", "switch", "while", "do-while", "for", "goto", "continue", "break", -->
<!ELEMENT statement      ((identifier, statement) | statement | %expression_opt;
                        | (symbol*, declaration*, statement*)
                        | (expression, statement, statement?) | (expression, statement)
                        | (statement, expression)
                        | (%expression_opt;, %expression_opt;, %expression_opt;, statement)
                        | identifier | continue | break)>
<!ATTLIST statement      %attr.common; %attr.rhs; %attr.control;>
<!-- rhs =  binops, "array", "funcall", "conditional", "cast",  unary ops, "paren",
          "sizeof-exp", "sizeof-type", "dot", "arrow", "identifier", "constant"-->
<!ELEMENT expression     ((expression, expression) | (expression, expression, expression)
                        | (type_name, expression) | expression | type_name | expression+
                        | (expression, identifier) | identifier | constant)>
<!ATTLIST expression     %attr.common; %attr.rhs; %attr.exp;>

<!-- terminals -->
```

```
<!-- rhs = identifier, enum_constant -->
<!ELEMENT  identifier    (#PCDATA)>
<!ATTLIST  identifier    %attr.common; %attr.rhs; %attr.ref;>
<!ELEMENT  typedef_name  (#PCDATA)>
<!ATTLIST  typedef_name  %attr.common; %attr.ref;>
<!-- string, char, int, float -->
<!ELEMENT  constant      (#PCDATA)>
<!ATTLIST  constant      %attr.common; %attr.rhs;>
<!-- empty_expression, for, e.g., 'for (;;);' -->
<!ELEMENT  empty_expression EMPTY>

<!-- the following are reserved words -->
<!ELEMENT  auto        EMPTY>        <!ELEMENT  break       EMPTY>
<!ELEMENT  char        EMPTY>        <!ELEMENT  const       EMPTY>
<!ELEMENT  continue    EMPTY>        <!ELEMENT  double      EMPTY>
<!ELEMENT  extern      EMPTY>        <!ELEMENT  float       EMPTY>
<!ELEMENT  int         EMPTY>        <!ELEMENT  long        EMPTY>
<!ELEMENT  register    EMPTY>        <!ELEMENT  short       EMPTY>
<!ELEMENT  signed      EMPTY>        <!ELEMENT  static      EMPTY>
<!ELEMENT  typedef     EMPTY>        <!ELEMENT  unsigned    EMPTY>
<!ELEMENT  void        EMPTY>        <!ELEMENT  volatile    EMPTY>
<!ELEMENT  ellipsis    EMPTY> <!-- for ... -->

<!-- DTD for ANSI C types -->
<!ELEMENT  type        (t_prim | t_enum | t_struct | t_union
                      | t_function | t_pointer | t_array | t_empty)>
<!ATTLIST  type         size             CDATA           #REQUIRED
                        id               ID              #IMPLIED
                        type_ref         IDREF           #IMPLIED
                        is_const        (true | false)   #REQUIRED
                        is_volatile     (true | false)   #REQUIRED>
<!-- primitive types -->
<!ELEMENT  t_empty       EMPTY>      <!ELEMENT  t_void        EMPTY>
<!ELEMENT  t_char        EMPTY>      <!ELEMENT  t_int         EMPTY>
<!ELEMENT  t_float       EMPTY>      <!ELEMENT  t_double      EMPTY>
<!ELEMENT  t_identifier  (#PCDATA)>  <!ELEMENT  t_constant    (#PCDATA)>
<!ATTLIST  t_char       is_signed       (true | false)   #REQUIRED
                        is_unsigned     (true | false)   #REQUIRED>
<!ATTLIST  t_int        is_long         (true | false)   #REQUIRED
                        is_longlong     (true | false)   #REQUIRED
                        is_short        (true | false)   #REQUIRED
                        is_signed       (true | false)   #REQUIRED
                        is_unsigned     (true | false)   #REQUIRED>
<!ATTLIST  t_double     is_long         (true | false)   #REQUIRED>
<!ELEMENT  t_prim      (t_void | t_char | t_int | t_float | t_double)>
<!ELEMENT  t_enum      (t_identifier, t_constant)*>
```

```
<!ATTLIST  t_enum       tag              CDATA           #IMPLIED
                        num              CDATA           #REQUIRED>
<!ELEMENT  t_field     (type)>
<!ATTLIST  t_field      name             CDATA           #IMPLIED
                        offset           CDATA           #REQUIRED
                        bit_offset       CDATA           #IMPLIED
                        bit_size         CDATA           #IMPLIED>
<!ELEMENT  t_struct    (t_field)*>
<!ATTLIST  t_struct     tag              CDATA           #IMPLIED
                        num              CDATA           #REQUIRED>
<!ELEMENT  t_union     (t_field)+>
<!ATTLIST  t_union      tag              CDATA           #IMPLIED
                        num              CDATA           #REQUIRED>
<!ELEMENT  t_function  (type, t_identifier?)+>
<!ATTLIST  t_function   is_ellipsis     (true | false)   #REQUIRED
                        is_old_type     (true | false)   #REQUIRED
                        arg_num          CDATA           #REQUIRED>
<!ELEMENT  t_pointer   (type)>
<!ELEMENT  t_array     (type)>
<!ATTLIST  t_array      num              CDATA           #IMPLIED>

<!-- DTD for the symbol table -->
<!ELEMENT  symbol       EMPTY>
<!ATTLIST  symbol       name             CDATA           #REQUIRED
                        type_ref         IDREF           #REQUIRED
                        ast_ref          IDREF           #REQUIRED
                        namespace       (normal | label | tag | member)    #REQUIRED
                        namelevel       (file | block)   #REQUIRED
                        is_tentative    (true | false)   #REQUIRED
                        is_incomplete   (true | false)   #REQUIRED
                        is_declaration  (true | false)   #REQUIRED
                        is_definition   (true | false)   #REQUIRED
                        is_global       (true | false)   #REQUIRED
                        is_enum_const   (true | false)   #REQUIRED
                        is_typedef      (true | false)   #REQUIRED
                        is_static       (true | false)   #REQUIRED
                        is_argument     (true | false)   #REQUIRED
                        is_builtin      (true | false)   #REQUIRED>
```

参考文献

[1] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Second Edition).* http://www.w3.org/TR//REC-xml.

[2] Greg J. Badros. *JavaML: A markup language for java source code.* http://www.cs.washington.edu/homes/gjb/JavaML/.

[3] ECMA (European Computer Manufacturers Association). *Portable Common Tool Environment (PCTE) - Abstract Specification*, 1997. ftp://ftp.ecma.ch/ecma-st/Ecma-149.pdf.

[4] Electronic Industries Association CDIF Technical Committee. *CDIF CASE Data Interchange Format - Overview, EIA/IS-106,* 1994. http://www.eigroup.org/cdif/.

[5] 権藤 克彦, 川島 勇人. *XCI (Experimental ANSI C interpreter) Homepage.* 北陸先端科学技術大学院大学 http://www.jaist.ac.jp/~gondow/xci/.

[6] Object Management Group (OMG). *formal/00-11-02 (XML Metadata Interchange (XMI) version 1.1).* http://www.omg.org/technology/documents /formal/xmi.htm.

[7] Rational Software Corporation. *Rational Rose.* http://www.rational.com/products/rose/index.jsp.

[8] GNU Project. *GCC.* Free Software Foundation. http://www.gnu.org/.

[9] M. Weiser. Program slicing. *IEEE Transaction of Software Engineering*, SE-10(4):352-357, 1984.

[10] WWW Consortium (W3C). *Document Object Model (DOM).* http://www.w3.org/DOM/.

[11] World Wide Web Consortium (W3C). *XSL Transformations (XSLT) Version 1.0.* http://www.w3.org/TR/xslt.

[12] World Wide Web Consortium (W3C). *XML Path Language (XPath) Version 1.0.* http://www.w3.org/TR/xpath.

[13] World Wide Web Consortium (W3C). *Namespaces in XML.* http://www.w3.org/TR/REC-xml-names/.

[14] World Wide Web Consortium (W3C). *Resource Description Framework (RDF) Model and Syntax Specification.* http://www.w3.org/TR/1999/REC-rdf-syntax/.

[15] World Wide Web Consortium (W3C). *Resource Description Framework (RDF) Schema Specification 1.0.* http://www.w3.org/TR/rdf-schema/.

[16] WWW Consortium (W3C). *W3C XML Schema.* http://www.w3.org/XML/Schema.

[17] INSTAC XML SWG. *RELAX (Regular Language descrip-tion for XML).* http://www.xml.gr.jp/relax/.

[18] World Wide Web Consortium (W3C). *Cascading Style Sheets, level 2 CSS2 Specification.* http://www.w3.org/TR/REC-CSS2/.

[19] World Wide Web Consortium (W3C). *Extensible Stylesheet Language (XSL) Version 1.0.* http://www.w3.org/TR/xsl/.

[20] World Wide Web Consortium (W3C). *HTML 4.01 Specification.* http://www.w3.org/TR/html401/.

[21] David Megginson. *SAX 2.0: The Simple API for XML.* http://www.megginson.com/SAX/index.html.

[22] World Wide Web Consortium (W3C). *XML Pointer Language (XPointer) Version 1.0.* http://www.w3.org/TR/xptr/.

[23] World Wide Web Consortium (W3C). *XML Linking Language (XLink) Version 1.0.* http://www.w3.org/TR/xlink/.

[24] World Wide Web Consortium (W3C). *XQuery 1.0: An XML Query Language.* http://www.w3.org/TR/xquery/.

[25] IBM. *alphaWorks : XML Parser for Java.* http://www.alphaworks.ibm.com/tech/xml4j.

[26] Sun Microsystems. *Java(TM) API for XML Parsing ("JAXP").* http://java.sun.com/xml/jaxp/index.html.

[27] The Apache Software Foundation. *Xerces Java Parser Readme.* http://xml.apache.org/xerces-j/.

[28] James Clark. *XP - an XML Parser in Java.* http://www.jclark.com/xml/xp/.

[29] Tim Bray. *An Introduction to XML Processing with Lark and Larval.* http://www.textuality.com/Lark/.

[30] S. Johnson. Lint, a C Program Checker, Unix Programmer's Manual. AT&T Bell Laboratories, 1978.

[31] University of Wisconsin. *The Wisconsin Program-Slicing Tools.* http://www.cs.wisc.edu/wpis/slicing_tool/.

[32] The Unravel Project. *The Unravel Program Slicing Tool.* http://hissa.nist.gov/unravel/

[33] Shigio Yamaguchi. *GNU Global-Source Code Tag System for C, C++, Java and Yacc.* ftp://ftp.gnu.org/gnu/global/global-4.1.tar.gz.

[34] Andrew M. Bishop. *The Cxref Homepage.* http://www.gedanken.demon.co.uk/cxref/.

[35] Organization for the Advancement of Structured Information Standards (OASIS). *The XML Cover Pages.* http://xml.coverpages.org/xml.html.

[36] Michael Kay. *What kind of language is XSLT?*. http://www-106.ibm.com/developerworks/xml/library/x-xslt/.

[37] IBM. *XML Parser for Java(XML4J)*. http://www.alphaworks.ibm.com/tech/xml4j.

[38] Sun Microsystems. *Java API for XML Processing (JAXP)*. http://java.sun.com/xml/jaxp/index.html.

[39] Steven P. Reiss and Manos Renieris. *Encoding program executions*. In Proc. 23rd Int. Conf. on Software Engi-neering (ICSE2001), pages 221-230, 2001.

[40] IEEE Computer Society's Task Force on Professional Computing Tools P1175. *A Standard Reference Model for Computing System Tool Interconnections*. IEEE Publications, 1991.

[41] Sun Microsystems. *Displaying a DOM Hierarchy (The Java(TM) Web Services Tutorial)*. http://java.sun.com/webservices/docs/ea1/tutorial/doc/JAXPDOM3.html.

[42] 鯵坂 恒夫. 開放型 CASE プラットフォーム. コンピュータソフトウェア, Vol.10, No 2, pp.4-12, 1993.

[43] 鯵坂 恒夫 沢田 篤史 満田 成紀. ソフトウェア評論 Emeraude PCTE. コンピュータソフトウェア, Vol.10, No 2, pp.65-77, 1993.

[44] 篠木 裕二, 西尾 高典, 吉川 彰弘. CDIF-CASE データ変換形式. コンピュータソフトウェア, Vol.10, No 2, pp.13-25, 1993.

[45] 福安 直樹, 山本 晋一郎, 阿草 清慈. 細粒度リポジトリに基づいた CASE ツールプラットフォーム Sapid. 情報処理学会論文誌, Vol.39 No.6, pp.1990-1998, 1998.

[46] 戸板 晃一, 山本 晋一郎, 阿草 清滋. XML を用いたソフトウェア関連文書とソースプログラムの整合性検査ツール. 日本ソフトウェア科学会 FOSE2001, pp.129-140, 2001.

[47] 有賀 寛朗, 山本 晋一郎, 阿草 清滋. ソフトウェア構造解析情報に基づくツールプラットホームシステム. 電子情報通信学会ソフトウェアサイエンス研究会, Vol.94, No.15, pp.25–32.

[48] 橋本 靖, 山本 晋一郎, 阿草 清滋. Program Slicing を利用したプログラムカスタマイザ. 電子情報通信学会ソフトウェアサイエンス研究会, Vol.94, No.10, pp.73–80 1994.

[49] 大橋 洋貴, 山本 晋一郎, 阿草 清滋. ハイパーテキストに基づいたソースプログラム・レビュー支援ツール. 電子情報通信学会ソフトウェアサイエンス研究会, Vol.98, No.28, pp.15–22 1998.

[50] 高田 智規, 佐藤 慎一, 飯田 元, 井上 克郎. ソースコード解析ツール開発支援システムの試用. 電子情報通信学会論文誌 D-I, Vol.J80-D-I, No.3, pp.317–318 1997.

[51] 磯田 定宏, 黒木 宏明. 統合化 CASE システム SoftDA の機能 —上流と下流の統合化コンピュータソフトウェア, Vol.10, No 2, pp.26-37, 1993.

[52] B.W. Kernighan and D.M. Ritchie 著, 石田 晴久 訳. プログラミング言語 C 第 2 版. 共立出版社, 1989.

[53] 下村 隆夫. プログラミングスライシング技術と応用. 共立出版, 1995.

[54] 鯵坂 恒夫, 佐伯 元司. 方法論工学と開発環境. 共立出版, 2001.

[55] B.W. Kernighan and R. Pike 著, 福崎 俊博 訳. プログラミング作法. ASCII, 2000.

[56] Mark Williams Company 編, 坂本 文, 今泉 貴史 監訳. ANSI C 言語大辞典. パーソナルメディア, 1990.

[57] A.V. Aho, R. Sethi, J.D. Ullman 著, 原田 賢一 訳. コンパイラ I –原理・技法・ツール–. サイエンス社, 1990.

[58] 佐々 政孝. プログラミング言語処理系. 岩波書店, 1989.

[59] 中山 幹敏, 奥井 康弘 編. 改定版 標準 XML 完全解説 上・下. 技術評論社, 2001.

[60] Natanya Pitts-Moultis, Cheryl Kitk 著, 山本 浩 訳. XML 実線ガイド 上・下. アスキー出版局 1999.

[61] 丸山 宏, 田村 健人, 浦本 直彦 著/訳. XML と Java による Web アプリケーション開発. ピアソン・エデュケーション, 1999.

[62] Neil Bradley 著, 安藤 慶一 訳. XSLT 完全活用マニュアル. ピアソン・エデュケーション, 2001.

[63] 浅海 智晴 *XML/DOM Programming*. 秀和システム, 2001.

[64] 古籏 一浩. HTML タグ辞典. 技術評論社, 2000.

[65] 権藤 克彦. Java によるプログラミング入門. サイエンス社, 2000.