修 士 論 文

実行系列を XML で表現した動的プログラムスライサの実装と評価

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

武田　洋佑

2003 年 3 月

修 士 論 文

実行系列を XML で表現した動的プログラムスライサの実装と評価

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

110073 武田　洋佑

提出年月: 2003 年 2 月

概 要

本稿では、プログラムの実行系列を XML で表現し中間データとして扱うことによって、動的スライサの実装を行い、開発効率を考慮した評価を行う。

より効率のよいソフトウェア開発には、動的解析が有効であるが、静的解析器に比べ、動的解析器の作成が困難である。なぜならば、静的解析であれば、AST+記号表+型情報+クロス参照とよく知られたデータ構造が存在するが、動的解析用の実行系列は、コンパイラの教科書や論文にもほとんど記述が無いため、どうモデル化や表現してよいか不明であるためである。これはスライシング技術にも当てはまる。また、一般的に、データの抽出から解析までを一貫して行う手法を取ると、データの再利用性に乏しい。これにより、字句解析器や構文解析器などのデータ抽出器を動的解析器ごとに作ることになる。これは、動的解析の本質でないにも関わらず、コストのかかる作業である。

本研究では、プログラムの実行系列に対し、動的スライサ用に動的情報 (変数の値やポインタ値の履歴等) を XML で表現したデータスキーマを定義し、中間データとして扱う。これに基づいた ANSI C サブセットを対象とした動的スライサの実装実験を行った。この実験に限れば、実行系列を XML 文書にすることによって、データ抽出部分と依存解析部分を分離ができ、開発コストの削減を確認できた。

目 次

表 目 次

図 目 次

第1章　はじめに

1.1　背景

スライシングの研究は主に実行効率を重視した研究であり、開発効率は軽視されている。プログラムスライシング技術を最初に考案したのは、1982年のMaryland大学のMark Weiserである。プログラムスライシングは一般的に静的スライシングと動的スライシングの2つに分けられる。Weiserのものは静的スライシングである。動的スライシングは、1988年に、KorelとLaski[4]が提案した。その後、1990年のAgrawalの研究[6]で、動的スライスの計算のためのいくつかのアプローチが提案された。これは、静的解析を拡張したものと動的依存グラフを使ったものである。前者は、非常に能率的だが、スライスが冗長である可能性があり、後者は、正確なスライスが求められるが、データサイズは実行系列の大きさに依存する。実行系列はループ文の実行回数に依存し巨大に膨れ上がることもある。この問題に、多くの研究者がデータサイズの軽減に関する研究を行っている。2001年のSteven P.Reissの研究[11]もその一つである。また、Pandeの研究[13]のように、動的解析でのプログラム言語のフルセット対応に関する研究も多い。Anthony M. Sloaneの研究[12]では、プログラムツリーに対してマーキングし、スライスの精度向上を目論んでいる。

動的スライサの開発コストは一般的に高いといわれている[32]。その理由として、個別の解析器が必要であり、実行環境が必要で、さらに、実行環境から実行系列を抽出するために、実行系列のモデル化が必要であるが、実行系列のモデルや表現の仕方は、コンパイラの教科書や、論文にもほとんど載っておらず、各々の開発者が決めなければならないためである。

XMLは、構造化文書を記述するためのマークアップ言語である。XMLの特徴として、プラットフォームに依存しないことや、新しい言語を生み出す拡張性、階層構造による高度なデータ記述力などがある。これらの利点は、動的スライシングで必要な実行系列の表現するのに有用であり、開発コストの削減に貢献すると考える。

XMLを利用して動的情報を細粒度で取得する研究は我々の知る限り存在しない。XMLをプログラムの解析に利用している研究として、JavaML[30]がある。JavaMLは、オブジェクトクラスまでをマークアップする。これは、粗粒度であり、スライシングに必要な情報をマークアップしていない。また、JAISTの川島と権藤の研究によるACML[31]や、Sapid[33]のI-modelは、識別子、変数、定数などの細かい情報までマークアップしていて細粒度である。よって、静的解析に必要な情報は十分である。しかし、これらの研究は全

て変数値やポインタアドレス、ループの回数など動的な情報を考慮した形式になっておらず、動的スライサの実行系列としては不十分である。

1.2 目的

本研究は、プログラムの実行系列を XML(Extensible Markup Language) で表現し、それを利用した動的スライサの実装と、開発効率を考慮した評価を行う。

DTD(Document Type Definition) を用いて適切に設計したデータスキーマによって、データ抽出と依存解析部分の分離を行い、データの再利用性の向上、容易なデータスキーマの拡張を考慮し、DOM(Document Object Model) などの XML 関連技術の利用で動的スライサの大幅な開発コスト削減が期待できると考える。

我々のゴールは、フルセットの ANSI C に対応した動的スライサの実装である。これには開発コストの削減は必要不可欠であり、開発効率の向上は動的スライシングの研究の加速に大きな役割を与えると考える。そこで、我々はこの研究の第一歩として、ANSI C プログラムのサブセットを用いた動的スライサの実装を行い、XML の有用性を評価する。

1.3 特色

本研究の特色は XML を実行系列の表現形式に利用し、動的スライサの開発コスト削減を狙う点である。

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、動的スライサに必要な実行系列の表現にも利点が多いと考える。その XML の主な利点を以下に示す。

- 階層構造による高度なデータ記述力
  構文情報などの木構造、実行の履歴などの表構造、依存グラフなどのグラフ構造など、さまざまなデータ構造を記述可能である。

- 複数の表示方法
  XSLT で HTML への変換や、XSL-FO、CSS を使っていろいろな表現が可能なため、動的スライサの結果をグラフィカルに見せることも可能である。

- 多彩なデータの検索方法
  XSLT や XPath を用いれば、DOM の単純な操作では煩雑になる XML 文書内の情報でも、強力なデータ検索能力で容易に検索が可能である。

- 豊富な関連技術
  さまざまな関連技術が存在し、それらのほとんどが安価で使用可能である。

本研究では、これらの XML の利点を使い、ANSI C プログラムの実行系列をマークアップするためのデータスキーマを定義する。データスキーマ定義のポイントは以下の通りで

ある。

- 情報取得の粒度
  全ての情報を細粒度で取得すると、サイズが大きくなり、構造も複雑になるので扱いにくくなる。しかし、粗粒度で取得すると動的スライシングに必要な情報を持っていない可能性が高くなる。よって、適度な粒度の設定が必要であるが、非常に困難な作業である。

- 抽象度の設定
  静的解析ツール用には、AST + 記号表 + 型情報 + クロス参照 と、よく知られたデータ構造が存在する。しかし、動的スライシングを含めた動的解析ツール用には、どうモデル化や表現してよいか、コンパイラの教科書や論文にもほとんど記述がない。よって、開発者自身が抽象度の設定をしなければならず、これは大きな労力である。

- 冗長な情報を持たない。
  実行系列は、実行された命令の延べ数に比例するため、サイズが大きくなりやすい。データサイズが大きくなれば、情報の操作も、理解も難しくなる。よって、できる限り、冗長な情報を省いて、データサイズを大きくならなくしたい。

- 構文情報の排除する。
  動的スライシングは、ほとんどの構文情報を必要としない。よって、構文情報は余分な情報である。上述したデータサイズの問題もあるため、不必要な情報は、できる限り排除する。また、構文情報は静的にも取得できるため、後から追加することも可能である。さらに、ACML[31] を使えば、細粒度で構文情報が利用できる。

- よく使う情報は、冗長であっても必要である。
  実行系列では、冗長性を省きたい。しかし、冗長性を省くことによって、毎回同じ手続きをしないと利用できないような場合がある。これはアプリケーションの開発コストとしてはマイナスである。多少、データサイズが大きくなっても、よく使う情報はアプリケーションから参照しやすく取得するべきである。

- 実行系列抽出側と解析側の意図
  実行系列は、動的スライスの中間データである。実行系列の抽出側開発者と、解析側開発者の意図は必ずしも同じであるとはいえない。抽出側は、プログラムの抽象構文木にしたがって、評価順序に添った形で、変数の情報を出力する方が楽である。しかし、解析側では、同じ種類の情報はまとまっていてくれた方が扱いやすい。例えば、ある式で、定義された変数と使用された変数は、まとまってくれていると扱いやすい。

また、XML の利点を使うことによる利点と欠点を以下に書く。

- 利点

  - 中間データがテキストファイルであるため容易に内容の確認をすることが出来る。
  - XML 文書にすることによって XML 用のさまざまな API が利用可能となり、ツールの本質でない部分の開発を分離することが可能である。
  - XML 文書はプラットフォームに依存しないため、例えば、データ抽出部分を Solaris 上の C 言語で開発し、依存解析部分を Windows 上の JAVA で開発するなど、開発言語に囚われない実装が可能となる。

- 欠点
  - XML 文書にすることで、タグ付けされるため、例えば CSV 形式で書いたデータよりサイズが大きくなる。
  - XML を利用することにより、データサイズが増え、テキスト形式であるため字句解析、構文解析に時間がかかるため実行速度は落ちる。

これらの利点より、開発効率の削減が可能であると考える。さらに、データスキーマを動的 CASE ツール用に拡張すれば、プロファイラーやプログラムシュミレータなどのデータとしても利用可能であり、動的 CASE ツールのプラットフォームとしての利用も期待できる。また、これら欠点は、現時点でハードウェア性能の前では無視できるほどと考える。

1.4 アプローチ

本研究では、XML を用いた動的スライサの実現の第一歩として ANSI C[1]のサブセットをターゲットとした動的依存解析ツールの実装実験を行った。我々の小規模な動的スライサ実装実験では、XML が動的スライサの開発コストの削減に有効であることを確認した。我々が考える動的スライサの構成要素は以下の通りである。

- XCI : Experimental C Interpreter
  ANSI C プログラムを実行し、実行系列を抽出するインタプリタ
- 動的 ACML : Dynamic ANSI C Markup Language
  ANSI C プログラムの実行系列用データスキーマ
- 依存解析ツール
  動的スライシングのアルゴリズムに基づいて動的依存関係を抽出する解析ツール

まず、XCI によって ANSI C プログラムからデータスキーマである動的 ACML にそった XML 文書として実行系列を抽出し、その XML 文書を XML パーサを通じ DOM や SAX などで、依存解析しスライスを抽出する。

[1] ISO/IEC9899-1990

研究の最初の作成実験として、2 通りの ANSI C サブセットを決め、特定のアルゴリズムの動的スライサを作成した。この実験は、XML の動的スライサの有用性を例題として適切であると考える。採用したアルゴリズムと、そのアルゴリズムに対する ANSI C サブセットの概要は以下の通り。

- Korel と Laski のアルゴリズム
  動的スライシングの最初のアプローチであり、基本的で有用である。これを基に XML の有用性を確認する。ANSI C サブセットは、スカラ変数と `while` 文、`if` 文の制御文からなる小規模なものである。システムコール、ライブラリ関数、関数呼び出し、ポインタ演算は行わない。ただし、特定のライブラリ関数は使用する。

- Agrawal のアルゴリズム
  より正確にポインタを計算するために考えられたアルゴリズムである。これを基に実用的な動的スライサの第一歩とする。ANSI C サブセットは、スカラー変数、配列、構造体、ポインタ変数、と制御文 (`while` 文、`if-else` 文) を使用する。関数呼び出し、システムコールは使わない。ライブラリ関数は特定のものだけ使用する。

1.5 結果

我々は、実装実験で、2 種類の依存解析ツールを、一人のツール開発者がそれぞれ約 2 週間で実装した。これより、我々の小規模な実装実験に限るが XML の開発コスト削減に対する有用性を確認した。開発コスト削減の要因を以下に示す。

- 中間データをテキストファイルで導入
  - 実行系列の抽出と解析の分離
    実行系列を中間データとして、テキストファイルである XML 文書に書き出した。これにより、我々の小規模な実験に限り、実行系列の抽出をする実行環境と、解析をする依存解析ツールの実装を分離できた。
  - テストケースの手作業での作成
    中間データがテキストファイルであるため、開発者が認識しやすい形となった。これにより、テストケースを開発者が手作業で書き出すことが可能となった。テストケースが手作業で作れると、実行環境が実装できて居なくても、依存解析ツールの動作確認が行える。
  - 実行系列を目で確認
    実行系列がテキストファイルであるため、テストケースの内容を確認しながら、依存解析ツールの実装を行うことができた。これは、バイナリ形式や、メモリ上に展開される実行環境一体型ではできない作業である。

- 実行系列を XML で表現

- 実行系列の構造に意味を付加
  XML 文書は構造化文書であるため、情報が付加されている。これにより、XML 文書内の情報の認識率を高めることができる。例えば、`<name>` タグがあれば、これは名前を表しているものだと、推測することが可能である。
- 字句解析、構文解析を XML 関連技術で補完
  実行系列の字句解析、構文解析器を実装するのは、動的スライサの本質的な内容ではないにもかかわらず、非常にコストが高い。しかし、実行系列を XML 文書で表すことよって、XML 用に用意された XML パーサが利用可能である。これは、多くの場合安価にで手に入る。
- DOM の導入による文書操作コストの削減
  DOM は、XML 文書操作でよく使われる手法である。DOM は機能が簡単なため習得に時間が少なくてすむ。また、DOM で使用する DOM ツリーは木構造であり、木構造は直感的に操作しやすい構造である。

実装実験で判明した改善点を以下にあげる。

- 動的 ACML の改善
  現在の動的 ACML は、Korel と Laski のアルゴリズムと Agrawal のアルゴリズムに特化した仕様となっている。さらに、その特化した中でも制限をして、取得する情報を絞っている。これにより、受理できるプログラムの命令が少なくなり、小さな規模のプログラムになってしまう。我々の考えているゴールである、フルセットの ANSI C には、おおくのライブラリ関数や、変数型、ポインタ操作などが存在する。これらの制限を改善することが必要と考える。

- XML 文書サイズ
  当初、我々はサイズ増加は無視できる範囲と考えていた。現在の動的 ACML では、ソースコードに対して、約 20 倍となっている。しかし、ANSI C の制限をはずすことによって情報量はさらに増えるため、節点集約法 [34] などの適用も考える必要がある。

- ライブラリの必要性、他の関連技術の導入
  実装実験では DOM を利用した。DOM は、単純な操作が多いため、技術の習得が容易であった。しかし、単純な操作であるがゆえに DOM では取得しにくい情報があった。例えば、「属性値に `reference` を持っている `variable` 要素の親要素である `line` 要素の `number` 属性値」などは、DOM では、アプリケーションで一つずつ操作を記述しなければならない。さらに、動的スライシングの操作ではこういった操作を繰り返すことが多い。そこで、定型作業をライブラリとして提供したり、XSLT や XPath の導入も必要と考える。また、XML 文書サイズの増加は、DOM ツリーをメモリ上に展開する DOM にも直接的に影響を与える。

1.6 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、動的スライシングは工学的に有用であり、XML が動的スライシングの開発コストに対して有用であることを述べ、研究の特色とその結果の概略を示した。

第２章 プログラムスライシングの概要と、主となる静的スライシング、動的スライシングのアルゴリズムの詳細を述べ、動的スライシングに関する研究の最近の動向と、プログラムスライシング技術の応用例を示す。

第３章 XML は、構造化データをテキストで形式で記述できる汎用フォーマットで、特に Web でのデータ交換用文書として注目を集めている技術である。本研究において、XML を中間データとして利用する。その XML 技術の概要と関連技術について述べる。

第４章 目標とする XML を利用した動的スライサの実装に対し、その一歩として、ANSI C のサブセットをターゲットに定め、第 2 章で紹介した動的スライシングのアルゴリズムを基とした実装のためのデータスキーマである動的 ACML の詳細を示す。

第５章 第 4 章で示した動的 ACML を利用し、第 2 章で紹介したアルゴリズムを基にした動的スライサの実装実験の実現について述べ、その結果を示す。

第６章 動的スライサの実装実験の結果から、開発コスト削減の要因を示し、この実験を通じて得た経験、XML や DOM などの関連技術の有用な点、不備な点などについて議論する。

第７章 本研究においてのまとめを行い、結論として、XML が動的スライサの開発コスト削減に寄与することを述べ、XML 技術が動的スライシングにおい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　プログラムスライシング

2.1　概要

プログラムスライシングとは、プログラムの意味を解析する技術の一つである。この技術は、1982年、Maryland大学のMark Weiser博士が初めて考案した[1]。最初はプログラムのデバッグを支援するために考えられていたが、現在では、ソフトウェアのテスト、デバッグ、改造、統合、リバースエンジニアリング、プログラム理解、メトリックス、コード最適化など、広範囲に応用されている技術である[35]。

プログラムスライシングはスライスを求める技術である。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オリジナルプログラムと同じ値の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スシング基準という。スライス基準において元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となる。よって、一つのプログラムの、一つのスライシング基準に対して、複数のスライスが存在する。また、もともとのプログラム自身もスライスである。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、スライスは可能な限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言うことができる。いかにして正確なスライスを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、オリジナルプログラムから、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。必要ない命令とは間接的にもデータ依存関係や制御依存関係が無い命令である。したがって、一般的に、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辿る事によって、スライスは、得られる。

スライシングにはスライシング基準から後ろに辿るものと、前に辿るものがある。前者を後ろ向きスライシングと言い、一般的にスライシングと呼ぶ場合、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る命令を抽出する。また、後者は前向きスライシングと言い、スライシング基準が影響を与える命令を抽出する。

一般的にスライシングは静的なものと動的なものに分けられる。これら二つの違いは入力を限定するかしないかである。前者は全ての入力を考慮し、後者は限定された入力のみを考慮する。一般的に前者を静的スライシング、後者を動的スライシングと呼ぶ。

静的スライシングと動的スライシングの詳細は以下の節で説明する。

2.2 静的スライシング

2.2.1 概要

静的スライシングとは、どんな入力を与えてもある変数に関してオリジナル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抽出する技術である。静的スライシングにより、プログラムの中からある機能を実現している部分だけを取り出すことができる。静的スライシングを行うには、どの命令に関して静的スライスを求めるか決める必要がある。これをスライシング基準とよぶ。また、静的スライシングによって抽出される成果物を静的スライスと呼ぶ。静的スライシングによってプログラムの理解が容易になり、テスト、デバッグ、保守などに利用される。静的スライシングの主な特徴は以下の通りである。

- スライシング基準において、任意の入力に対して元のプログラムと同じ動作をする。
  静的スライシングは、スライシング基準の変数に対して、依存の可能性を抽出する。静的スライシングは、プログラムのソースコードに対して行われる。よって、プログラムの分岐命令や配列のインデックスなどは、スライスの計算時に決定しないため、どんな入力値の場合でも、スライス基準において、同じ値を計算するスライスが抽出される。

- 静的な情報に対して、依存関係の可能性を計算するので、探索範囲が広くなる。
  静的スライシングでは、入力値を考慮しない。これにより変数の値など、動的に決定できるものが利用できない。これにより、スライスの探索範囲が広がる。例えば、if 文では、プログラム実行時に必要なのは、必ず真か偽のどちらか一方だけである。しかし、静的スライスを計算する段階ではこれは決定できず、両方を計算しなければならない。

- 正確なスライスを求めるのは困難
  上記二項目の理由と同じように、静的スライシングでは、入力値を考慮しないため、依存の可能性をすべて計算する必要がある。例えば配列 a がある場合、a[i] が参照され、a[j] が定義されていたとする。このとき、i と j の値が決まらないため、a[i] が参照されたとき、配列 a 全てが依存すると考えなければならない。よって、a[i] と a[j] の間にはデータ依存ができてしまう。これにより、実際、プログラム実行時には必要のない命令もスライスに含まれる可能性があり、一般にスライスが大きくなる。

- 実現は動的スライサに比べて容易
  静的スライシングは、実行環境を必要としない。これにより上記項目のデメリットが発生する。しかし、実行環境を必要とせず、静的情報のみに対して、スライシングを行うため必要とする情報量は動的な場合に比べ明らかに少なく、実装は動的スライサに比べ容易となる。

2.2.2 Weiser のアルゴリズム [1][2]

以下に Weiser の静的スライシングのアルゴリズムを示す。

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のノードに対して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向グラフがフローグラフである。

プログラムの定義 プログラムは次の文 (statment) からなると考える。

- 代入文
  - $x = y + z;$
  - $a[i] = 3;$
  - $b = a.c;$
- 分岐文
  - *if* (式){ 文の列 } *else* { 文の列 }
  - *if* (式){ 文の列 }
- ループ文
  - *while*(式){ 文の列 }
  - *for*(式; 式; 式){ 文の列 }
- 入力文
  - $c = fgetc(stdin);$
  - $scanf(\text{"}\%d\text{"}, \&a);$
- 出力文
  - $printf(\text{"}\%d\text{"}, c);$
  - $fputc(c, stdout)$

また、以下を総称して命令 (Instruction) と呼ぶ。

- 分岐命令
  分岐文の条件式の部分
- ループ命令
  ループ文の条件式の部分

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

図 2.1: サンプルプログラム 1

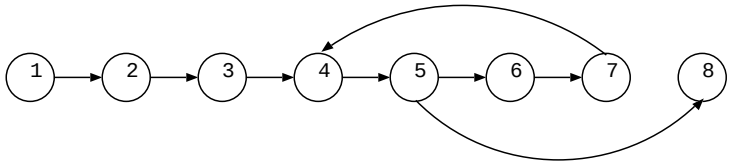

図 2.2: サンプルプログラム 1 に対するフローグラフの例

- 代入命令
  代入文
- 入力命令
  入力文
- 出力命令
  出力文

これら命令の間の制御の流れる方向を表した有向グラフがフローグラフになる。

フローグラフの定義 プログラム P のフローグラフ G=(N, A, e) の定義は以下の通り。

1. N はノードの集合で、ノードはプログラム内の命令からなる。
2. A はアークの集合で、アーク $(s,\ t)$ は、命令 s を実行したあと、制御が直ちに命令 t に移る可能性のあることを示す。
3. e はノードで、プログラム P を実行するときに最初に制御が移される命令である。

例として、階乗の計算をするプログラム図 2.1 のフローグラフを図 2.2 に示す。

データ依存関係： Data Dependence Relation

ある変数 w が存在するとき、命令 s で定義された変数 w が、w を使用している命令 t に到達するとき、「命令 s から命令 t に対してデータ依存関係 $DD(s,t)$ がある」という。ここで、「定義」とは、ある命令 s で変数 w の値が書き換えられたことをい

い、「使用」とは、命令 t で変数 w の値を参照したことを言う。また、「到達する」とは、命令 s が変数 w を定義し、かつ、フローグラフ上で命令 s から命令 t に至るパスが存在し、そのパス上で変数 w が再定義されないことを言う。つまり、命令 s で定義した変数の値を命令 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： Control Dependence Relation

命令 s が分岐命令またはループ命令で、命令 t がその命令の内部に直接含まれているとき、「命令 s から命令 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、命令 s と命令 t の間に新たな分岐命令またはループ命令が存在しないことを意味する。つまり、制御依存関係があるということは、命令 t の実行の可否が命令 s の実行結果に依存していることを意味する。

方法

静的スライシングのスライシング基準 `C = (u, V)` はプログラム内の命令 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。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、次のとおり定式化できる。

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、$A^i =$ {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. $SS = \bigcup_{i=0}^{\infty} A^i \cup \{u\}$

2.3 動的スライシング

2.3.1 概要

動的スライシングとは、ある入力を与えてプログラムを実行した場合に、その同じ入力に関しては、同じ値を計算する実行可能な部分プログラムを抽出する技術である。動的スライシングで抽出される成果物を動的スライスと呼び、動的スライシングが解析対象とする情報の列を実行系列と呼ぶ。動的スライシングは、プログラムの理解、コード最適化、デバッグ、ソフトウェアの自動テストなど幅広い分野に応用可能である。ここで、動的スライシングで扱う実行系列 (Execution History) とは、入力を与えてプログラムを実行した場合の、実行された命令の列を呼ぶ。また、実行系列で、p 番目に実行された命令の場所を、実行時点 p と呼ぶ [35]。動的スライシングの主な特徴は以下の通りである。

- スライシング基準において、特定の入力でのみ、元のプログラムと同じ値を計算する。
  動的スライシングは、スライシング基準に入力値を取り、その入力値に対してプロ

グラムを実行して抽出される。これにより、分岐命令のパスは特定され、配列のインデックスは固定される。例えば、

```
if(x < y)
   {x = 1;}
else
   {x = 2;}
```

とあるとき、入力値が $(x = 1, y = 2)$ でスライシングを行うと、else 部はスライスに含まれない。よって、このスライスは特定の入力に対してのみ同じ値を計算することとなる。

- 探索範囲を著しく限定している
  動的スライシングは、実行系列に対して解析を行う。実行系列は、プログラムを実行したときに発生する。この実行系列は、プログラムの入力値によって、変化する。これにより、一つの入力値で解析できる範囲は限定される。

- 正確なスライスを求めやすい
  動的スライシングは、入力値を取り、プログラムを実行して解析するため、変数の値が決まる。これにより、スライスは可能性ではなく、実際の依存があるパスが抽出される。例えば、配列 a、整数型 i、j が存在したとき、静的解析では a[i] と a[j] が依存してる可能性がある。しかし、動的解析では、i と j の値が決定しているので、i = j の時以外、依存がないことがわかる。また、制御命令の場合も変数の値が決まることによって実行されるパスが判明する。例えば if 文の条件式が真のとき、偽の場合に実行される命令群は、必ず実行されないため動的スライスには含まれない。また、ポインタ値が決まるため、エイリアスに正確に対応することもできる。

- 動的スライサの実現は静的スライサに比べて困難
  動的スライサの実現は、静的スライサの実現に比べて困難である。その理由として以下があげられる。

  - 実行系列を作り出すための実行環境が必要である。
    動的スライシングを行うためには、実行系列が必要である。実行系列を作り出すためには、プログラムの実行環境が必要である。GCC などの一般的な C 言語の実行環境では、実行系列を抽出できるようには設計されていない。このため、動的スライシング専用の実行環境が必要となる。この実行環境にかかる開発コストは少なくない。
  - 実行系列を扱うためのデータスキーマが必要である。
    静的スライシングはソースコードを基に解析をするため、全ての情報を保持することができる。しかし、動的スライシングは実行系列を基にする。実行系列

とはプログラムを実行したときの情報であるため、全ての情報を取得することはデータの種類や量が多く、一般に使うモデルや表現方法が決まっていないため、非常にコストがかかる作業であり。大変難しい。実行系列を抽出するためには、どの種類の情報を、どの粒度で取得するかを決定しなければならない。例えば、変数のシンボル名、値、型など情報の種類は多様であるが、動的スライシングに必要な情報は一部のみである。

- 動的スライシングに直接関係の無い部分の開発コストがかかる。
  動的スライサには実行環境が必要である。実行環境からデータを取得し、この情報を動的スライシングで使える実行系列としなければならない。このとき、字句解析、構文解析など、動的スライシングに直接関係の無い部分の開発コストが高い。
- 実行系列のデータサイズが大きくなる傾向がある。
  動的スライシングは、実行系列を解析する。実行系列はプログラムを実行して抽出される。この実行系列は、入力値によって、さまざまなサイズになる。例えば、30 行のプログラムがあったとき、25 行が `while` 文に依存していたとする。このとき、`while` 文が 1 回も実行されなければ、実行系列は、実行された 5 行から構成される。逆に、`while` 文が 100 回実行された場合、`while` 文に依存している 25 行も 100 度実行されるため、実行された 2500 行以上から実行系列は構成される。一般的に `while` 文などのループ命令は複数回実行される。実行系列のデータサイズは実行された命令の述べ数に比例して大きくなる。

2.3.2 Korel と Laski のアルゴリズム [4][5]

1988 年に Korel と Laski によってプログラムスライシングの動的なアプローチが考えられた。これを動的スライシングと呼ぶ。動的スライシングは、ある入力を与えたとき、ある変数に関しては元のプログラムと同じ動作をする部分プログラムを抽出する技術である。動的スライシングを行うには、まず、どの実行時点において、どの変数に関して動的スライスを求めるかスライシング基準を決める必要がある。動的スライスを求めるためのスライシング基準は以下の通りである。
プログラム P のスライシング基準 `C = (x, r, V)` とは、次の 3 つ組をいう。

1. x はプログラム P に与えた入力
2. r はプログラム P に入力 x を与えたときの実行系列における実行時点
3. V はプログラム P 内の変数の部分集合

以下に、Korel と Laski のアルゴリズムを示す。

```
 1 : GetIntArray(n,A);
 2 : min = A[0];
 3 : max = A[0];
 4 : sum = A[0];
 5 : i = 1;
□6 : while(i <= n){
 7 :    if(max < A[i])
 8 :       max = A[i];
 9 :    if(min > A[i])
10 :       min = A[i];
11 :    sum += A[i];
12 :    i++;
     }
13 : printf("min = %d\n", min);
14 : printf("max = %d\n", max);
15 : printf("sum = %d\n", sum);
```

図 2.3: サンプルプログラム 2

実行時点	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
行番号	1	2	3	4	5	6	7	9	10	11	12	6	7	8	9	11	12	6	13

表 2.1: 実行時点と行番号の対応の例 1

定義

動的スライスは、スライシング基準となる実行時点を基点とし、スライシング基準となる変数に関して、プログラムの実行系列上での命令の間のデータ依存関係と制御依存関係を逆向きに辿ることによって取り出される実行時点で実行された命令の集合として求めることができる。さらに、命令同一関係を導入したものを Korel の動的スライスと呼ぶ。

実行系列とは、入力を与えてプログラムを実行したときに実行された命令の履歴を指す。また、実行時点とは、実行系列上で n 番目に実行された命令のことで実行時点 n という。さらに、実行時点 n で実行された命令を Ins(n) で表す。配列の要素の最大値と最小値、合計を求めるプログラム (図 2.3) に対し、入力"n = 3, A = (2, 1, 3)"を与えて実行したときの実行時点と行番号の対応はは表 2.1 となる。

データ依存関係： Definition Use Relation

実行時点q で参照される変数w が存在し、実行時点p が実行時点q より前で、最後に変数w を定義しているとき、「実行時点p から実行時点q にデータ依存関係DU(p,

q) がある」という。データ依存関係があるとは、実行時点p で定義された変数w を実行時点q で参照したことを意味している。

制御依存関係： Test Control Relation

実行時点q で実行された命令Ins(q) へ静的制御依存関係[1]がある命令を持つ実行時点の中で、実行時点q 以前で最後に実行された命令持っている実行時点p が存在するとき「実行時点p から実行時点 q に制御依存関係TC(p,q) がある」という。制御依存関係があるとは、命令Ins(q) が実行されたことが命令Ins(p) の実行結果に依存していることを意味する。

命令同一関係： Identity Relation

実行時点p の命令Ins(p) と実行時点q の命令Ins(q) が存在して、Ins(p) = Ins(q) かつp < q のとき「実行系列p と実行系列q の間には命令同一関係IR(p,q) がある」という。つまり命令同一関係IR(p,q) とは、実行時点p と実行時点 q の違う実行時点で同じ行の命令が実行されたことを示している。

方法

動的スライシングのスライシング基準 C = (x, r, V) は、プログラムに与えられた入力x、プログラムに入力x を与えた時の実行系列上の実行時点r、および、プログラム内の変数の部分集合V によって与えられる。スライシング基準 C = (x, r, V) に関する動的スライス DS の求め方は、次のように定式化できる。

1. A^0 = { 実行時点 q| ある変数 $v \in V$ に対して、$q = Def(r,\ v)$、あるいは $TC(r,\ v)$}
2. $i \geq 1$ に対して、A^i = { 実行時点 p| ある実行時点 $q \in A^{i-1}$が存在して、$DU(p,\ q)$ もしくは $TC(p,\ q)$ もしくは $IR(p,\ q)$}
3. $DS = \{Ins(p) | p \in \bigcup_{i=0}^{\infty} A^i \cup \{r\}\}$

2.3.3 Agrawal のアルゴリズム [6][7]

1990 年、Agrawal は、動的スライシングの新しい方法として、静的スライシングを拡張したアプローチと、実行系列を使うアプローチを提案した。1991 年、Agrawal はポインタ解析を考慮した動的スライシングを提案した。これは、スライシング基準に変数のシンボル名ではなく、Cell と呼ばれるものを用いる方法である。Cell とは、(addr,len) の組で表され、addr はアドレス番地を、len はサイズを表す。この Agrawal の動的スライシングは、フローグラフより、データ依存関係と制御依存関係を計算することによって動的依存グラフ (DDG[2]) を作り、スライシング基準から DDG のパスを辿って動的スライスを取

[1] Weiser のアルゴリズムにおいての制御依存関係
[2] Dynamic Dependence Graph

```
 1 : scanf("%d", n);
 2 : z = 0;
 3 : i = 1;
 4 : while(i <= n){
 5 :     scanf("%d", x);
 6 :     if(x < 0)
 7 :         y = F1(x);
         else
 8 :         y = F2(x);
 9 :     z = F3(z, y);
10 :     i++;
     }
11 : printf("%d", z);
```

図 2.4: サンプルプログラム 3

実行時点	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
行番号	1	2	3	4	5	6	7	9	10	4	5	6	8	9	10	4	11

表 2.2: 実行時点と行番号の対応の例 2

り出す。以下にその定義と方法を示す。

定義

実行系列 実行系列の定義は Korel のアルゴリズムと同じである。例として、図 2.4 のプログラムを入力値"$n = 2$、$x = [-4, 3]$[3]"で実行したときの実行時点と行番号の対応 (表 2.2) を示す。
実行系列 hist は、実行時点 v_i の列であり、
$hist =< v_1, v_2, v_3 \ldots v_n >$
と表す。また、
$Last(hist) = v_n$
$Prev(hist) =< v_1, v_2, v_3 \ldots v_{n-1} >$
とする。さらに、$LastOccur(v, hist)$ は、行番号 v が実行系列 $hist$ で最後に実行された実行時点を返す。

使用と定義 変数 x を定義するとは、変数 x に値を設定することを言う。また、変数 x を使用するとは、変数 x の値を参照することをいう。命令 s に対して $def(s)$ は命令

[3] $x = [-4, 3]$ は、x の一度目入力で -4 を入力し、二度目の入力で 3 を入力することを表す

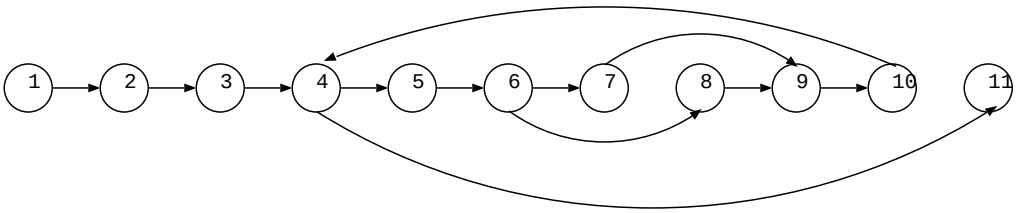

図 2.5: 実行系列２に対するフローグラフの例

s で定義される変数、$use(s)$ は命令 s で使用される変数の集合を表す。

フローグラフの定義　プログラム P のフローグラフ $G = (V, A)$ の定義は以下の通り

1. V はノードの集合で、ノードは実行系列上の実行時点からなる。
2. A はアークの集合で、アーク (s, t) は、命令 s を実行したあと、直ちに命令 t が実行されたことを示す。

例として表 2.2 の実行系列におけるフローグラフを示す。

Cell の交差： Cell Intersection

Agrawal の動的スライシングでは、変数のシンボル名を使わず、変数のアドレス値とサイズによって解析を行う。$CI(useCell, defCell)$ が真とは、ある実行時点 q に $useCell = (useaddr, usesize)$、$q > p$ である実行時点 p に $defCell = (defaddr, defsize)$ の二つの Cell が存在して、

- 完全一致
  $(defaddr \leq useaddr) \wedge ((useaddr + usesize) \leq (defaddr + defsize))$
- 前方交差
  $(defaddr \leq useaddr) \wedge ((defaddr + defsize) < (useaddr + usesize))$
- 後方交差
  $(useaddr < defaddr) \wedge ((useaddr + usesize) \leq (defaddr + defsize))$
- 一部交差
  $(useaddr < defaddr) \wedge ((defaddr + defsize) < (useaddr + usesize))$

のいずれかであるときである。CI が真で、かつ、前方交差、後方交差または一部交差のときに、$PreCell$ と $PostCell$ を定義する。これは、完全一致以外の場合、$useCell$ は、$defCell$ が定義しているのは一部であり、残りの部分を定義している別の $Cell$ が存在することを意味する。$PreCell(usecell, defcell)$ は、$usecell$ の範囲で、$defcell$ が定義してない部分を返す。$Postcell$ も同様である。前方交差には、$Postcell$ が存在し、後方交差には、$Precell$ が存在する。また、一部交差には、$Postcell$ と $Precell$ の両方が存在する。

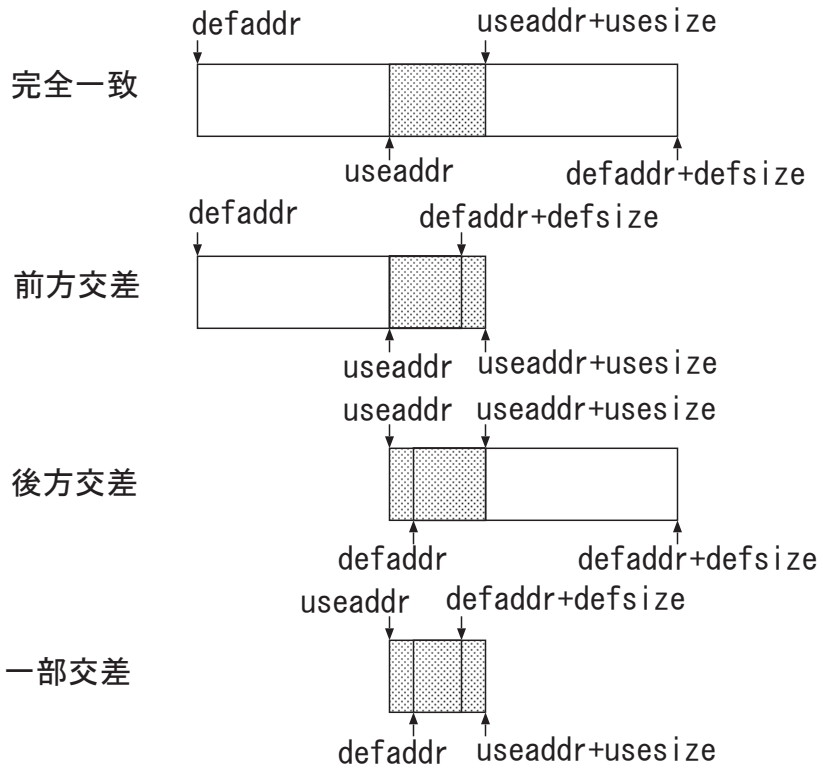
完全一致
defaddr
useaddr+usesize
useaddr
defaddr+defsize
前方交差
defaddr
defaddr+defsize
useaddr
useaddr+usesize
後方交差
useaddr
useaddr+usesize
defaddr
defaddr+defsize
一部交差
useaddr
defaddr+defsize
defaddr
useaddr+usesize

図 2.6: Cell の交差

動的到達定義： Dynamic Reaching Definitions

動的到達定義とは、cell が実行系列 hist で最後に定義された実行時点を表す。動的到達定義 DRD(cell, hist) は以下のとおり定式化される。

$$
\begin{aligned}
&DRD(cell,\ <>)\ =\ \phi\\
&DRD((addr,\ 0),\ hist)\ =\ \phi\\
&DRD(cell,\ hist)\ =\ if\ cell'\ =\ \phi\\
&\qquad\qquad then\ DRD(cell,\ Prev(hist))\\
&\qquad elseif\ CI(cell,\ cell')\\
&\qquad\qquad then\ \{Last(hist)\}\\
&\qquad\qquad\quad \cup DRD(Precell(cell,\ cell'),\ Prev(hist))\\
&\qquad\qquad\quad \cup DRD(Postcell(cell,\ cell'),\ Prev(hist))\\
&\qquad\qquad else\ DRD(cell,\ Prev(hist))
\end{aligned}
$$

ここで、cell' は、def(Last(hist))) である。

データ依存関係　データ依存関係とは、ある実行時点で使用された Cell の集合が、どの実行時点で定義されたかを求めた集合である。データ依存関係は以下のとおり定式化できる。

$$D(hist)\ =\ \bigcup_{\substack{cell\ \in\ use(Last(hist))\\ x\in\ DRD(cell,\ Prev(hist))}} \{(Last(hist),\ x)\}$$

制御依存関係　制御依存関係とは、実行時点 p で実行されている命令に対して、静的制御依存関係のある命令の中で、p よりも前に実行された時点で、一番 p に近いものを表す。制御依存関係は以下のとおり定式化できる。

$$C(hist)\ =\ \bigcup_{\substack{x\ \in\ ControlPred(Last(hist))\\ y\in\ LastOccur(x,\ Prev(hist))}} \{(Last(hist),\ y)\}$$

このとき、ControlPred(p) とは、ある実行時点 p が静的制御依存のある行数を表す。

動的依存グラフ： Dynamic Dependence Graph

動的依存グラフとは、実行系列の実行時点をノードとして、データ依存関係と制御依存関係を示したグラフである。動的依存グラフ DDG(V,A) は次のとおり定式化できる。

$$
\begin{aligned}
&DDG(<>) = (\phi, \phi)\\
&DDG(hist)\ =\ DDG(< Prev(hist)\ |\ Last(hist)\ >)\\
&\qquad\qquad =\ (V'\ \uplus\ Last(hist),\ A'\ \cup\ D\ \cup\ C)
\end{aligned}
$$

このとき、$DDG(Prev(hist))\ =\ (V',\ A')$ で、$\uplus$ は直和を表す。動的依存グラフの例として、図 2.5 の実行系列に基づいた DDG を示す。実線はデータ依存関係を、破線は制御依存関係を表す。

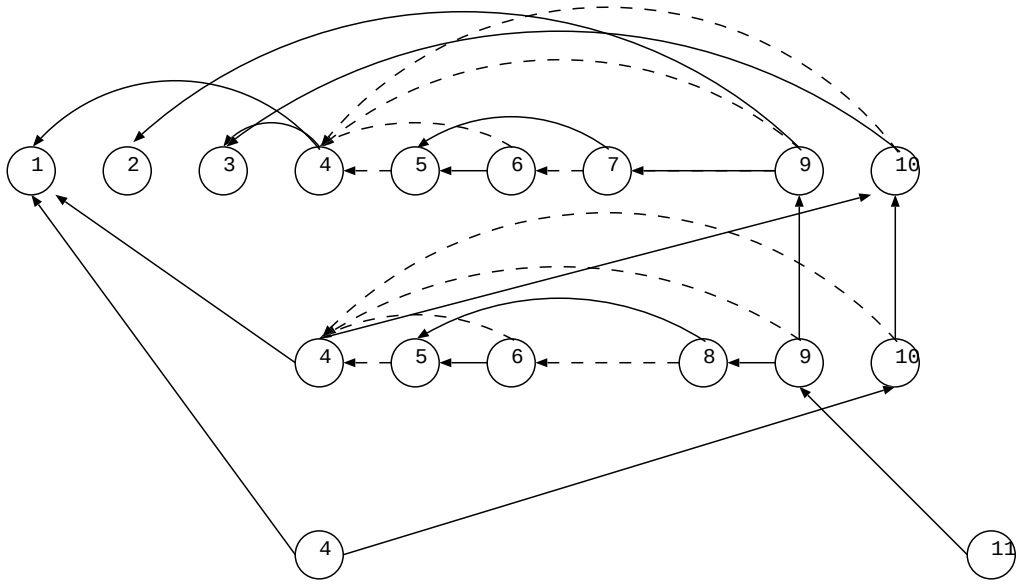

図 2.7: 動的依存グラフの例

方法

動的スライシングのスライシング基準$C = (x,\ r,\ V)$は、プログラムに与えた入力x、プログラムに入力xを与えた時の実行系列上の実行時点r、および、プログラム内の変数の部分集合Vによって与える。スライシング基準$C = (x,\ r,\ V)$に関する動的スライスDSの求め方は、次のとおり定式化できる。

$$DS(hist, cell) = \bigcup_{x \in DRD(cell, hist)} RN(x, DDG(hist))$$

RN：動的依存グラフで到達可能なノード

2.4 応用

スライシング技術は、ソフトウェアのテスト、デバッグ、改造、統合、リバースエンジニアリング、プログラム理解、メトリックス、コード最適化など、広範囲に応用されている技術 [8] である。その中で主なものを以下に説明する

2.4.1 デバッグ

スライシングは、元々デバッグを目的に考案された技術 [1] である。予期しない値を持った変数に関してスライシングすることにより、その変数に予期しない値を代入する要因の可能性がある命令を抽出することが可能である。静的スライシングを用いれば、オリジナ

ルプログラム内のエラーを起こしている可能性のある部分を抽出可能である。また、動的スライシングを用いれば、エラーを起こしている部分を含んだ実行系列を抽出でき、それを基にトレースすることによって、プログラム状態を再現可能である。また、静的スライシングの変形でデバッグのために考えられたダイシング[4][3][35] を用いれば、さらに効率を良くすることができる。

2.4.2 テスト

プログラムの品質向上には、十分なテストを行うことが必要である。プログラムスライシングは、プログラムのテストにも応用ができる。例えば、構造テストには、全ての分岐をテストするブランチテスト手法、全てのパスをテストするパステスト手法、変数が定義されてから使用される間のパスをテストするデータフローテスト手法などがあるが、ブランチテストでは各々の分岐部分でのスライスを抽出することにより分岐命令に関係する命令を抽出できる。パステストではスライスを抽出することにより、どこのパスを通ってないかが判明する。データフローテストはスライシングのデータ依存そのものである。スライシングはテストの希望通りのパスを通るように入力値の変更のために使われる [35]。

2.4.3 再利用

プログラムは数千行、数万行と言ったものも珍しくはない。しかし、その一部の記述は以前に書かれたプログラムの一部と同じであることも少なくない。その共通なコードを再利用できれば、プログラミングの効率化につながる。もし、プログラムの特定の機能を実現している部分のみを取り出すことができれば、プログラムの再利用は効率的に行われる。スライシング技術を用いることによって、特定の機能を実装しているプログラムの一部を正確に抽出することが期待できる。

2.4.4 統合

プログラムの再利用は頻繁に行われている。あるプログラムから取り出したある機能を持ったプログラムの一部分を別に統合することを考える。統合したプログラムでは、追加した機能部分が他の部分に影響を与え予期しない動作をしないこと、そして、追加した機能部分が他の部分に影響を受けて予期しない動作をしないことが要求される。そこで、統合したプログラムに対し、スライシングをすることによって、統合した部分の独立性を確認することが可能である。

[4]スライシングを別の基準で二回以上することにより、その差を解析し、より正確な分析をする技術。例えば、ある正常な値を計算する変数 x と、ある間違った値を計算する変数 y が存在するとき、x と y についてスライシングを行い、x についてのスライスに含まれる部分を、y に付いてのスライスから取り除くと、要因となっている点を絞れる。これをダイシングという。

第3章　XMLの概要と位置付け

本研究では、XML を動的スライサの実行系列を中間データとして表すために用いる。この章では XML の概要と特徴、関連技術について述べる。

3.1　XML(Extensible Markup Language)の勧告

XML は、World Wide Web コンソーシアム (W3C)[14] 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 は以下の設計目標に基づいている [15]。

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 と互換性を持つ。
4. XML 文書を処理するプログラムは容易にかける。
5. XML では、オプションの機能はできるだけ少なくし、理想的には一つも存在しない。
6. XML 文書は、人間に読みやすく、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ない。

XML は、SGML[1] と HTML[16] を基本とし、双方の利点を併せ持ち、かつ、欠点を補完するように設計されている。

SGML は ISO[2] によって 1986 年に制定された文書記述言語である。しかし、SGML は大規模で大変複雑なため、使用に対するコストが極めて高いもので、あまり普及していない。

一方、HTML[3] は、World Wide Web の普及とともに 1990 年ごろから表示用として広く普及した。HTML は SGML のサブセットであり、`<br>` や `<table>` などのタグを定義してある。一般的に Web ブラウザを通して情報を視覚的に伝えることを目的としている。HTML が普及した主な理由は、その文法が非常に簡単であることが大きな要因である。しかし、そのため HTML は拡張性の無い言語仕様となった。

[1] Standard Generalized Markup Language
[2] International Organization For Standardization(国際標準化機構)
[3] Hyper Text Markup Language

XML は、HTML と同様、SGML のサブセットとして設計された言語であり、SGML 同様、任意の種類のデータをマークアップすると言う目的は同じであるが、SGML の複雑な部分を可能な限り排除し、HTML の手軽さを継承している。以下に、XML が、HTML と SGML から継承した主な利点を挙げる。

- SGML から継承した利点
  - 任意の種類のタグを定義可能である。
  - 構造化文書の作成に適している。
  - DSSSL[4]などの強力な汎用スタイルシートの利用が可能である。
  - 厳密な文法チェックが可能であり、電子化が容易に行える。
- HTML から継承した利点
  - フォーマットが容易で読みやすい
  - ハイパーリンクを持っている
  - ウェブ上で配信、受信、処理が可能である。

3.2 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 の特徴を以下に挙げる。

3.2.1 データの構造化

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の交換が可能である [36]。また、XML は、リンクを持っている。これによって、一般的なデータ構造 (木構造、グラフ構造、リスト構造、表構造など) を柔軟に記述できる。

例えば、図 3.5 の様な表計算用のデータがあるとき、これをアプリケーション専用の形式で保存し、誰かに送れば、相手先は同じアプリケーションが無ければファイルを開くことすらできない。また、このデータを図 3.6 の様に CSV 形式とした場合、CSV は、図にあるようにカンマで区切られたテキストファイルである。これでは、情報は伝えられるがデータの意味が明確ではない。これを簡単な XML で表現すると図 3.7 となる。簡単な構造であるが、データの意味を明確に表すことができている [37]。

[4]Document Style Semantics and Specification Language

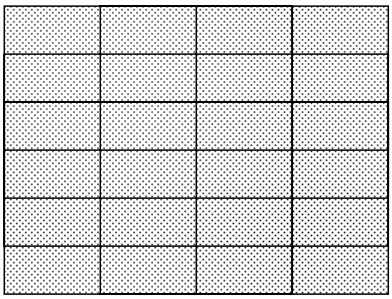

図 3.1: 表構造

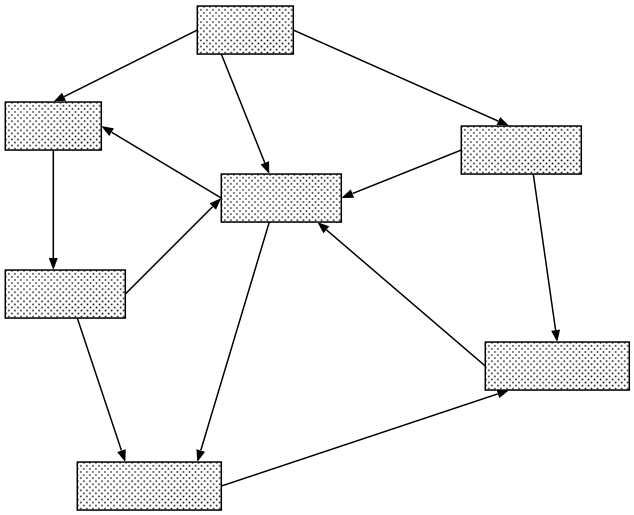

図 3.2: グラフ構造

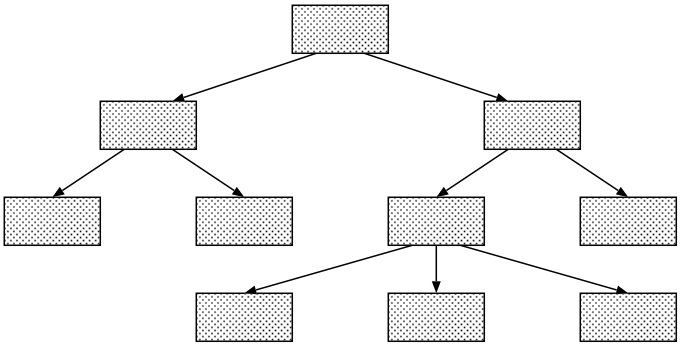

図 3.3: 木構造

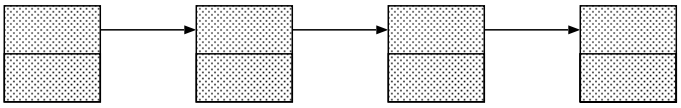

図 3.4: リスト構造

Index	Name	Price
058	CD-RW	15800

図 3.5: 2 行 3 列の表

```
Index,Name,Price
058,CD-RW,15800
```

図 3.6: CSV 形式

```
<item>
   <Index>058</Index>
   <Name>CD-RW</Name>
   <Price>15800</Price>
</item>
```

図 3.7: XML 形式

3.2.2 データの独立性

XML は、データの独立性が確立されている [38] 。W3C の勧告である XML は、ベンダーやプラットフォームに依存しない。W3C のメンバーにはどんな企業、団体や個人でもなることができる。メンバーであれば標準化のプロセスに参加することができる。これによって W3C の勧告は、相互運用性を備えたもので標準として多くの場面で利用される可能性が増える。また、この勧告に基づくことによって、他者の開発したアプリケーションとの相互性を持たせることが可能である。

3.2.3 一つのデータソースに対する複数の表示

XML は、いくつかのデータ表示方法を持っている。一般的に使用されるものは以下の 3 つである。

- CSS[5][17] でスタイルを定義し表示する。
- XSLT[20] で HTML に変換し表示する。
- XSLT で XSL[6][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。

これらの機能を利用することにより、一つの XML 文書を目的に合った表示方法で確認することができる。

[5] Cascading StyleSheet
[6] The Extensible Stylesheet Language

3.2.4 データ検索能力の強化

XML 文書には構造化されたデータが保存されている。しかし、これを効率的に利用するためには、効率的なデータの検索が必要不可欠である。そこで、XML では、関連技術である XSLT[20] や XPath[21] を使ってパターンマッチングを行うことによってデータの検索をすることが可能である。

3.2.5 より簡単なアプリケーションの開発

XML の親にあたる SGML は、マークアップと呼ぶタグの入れ子構造により、ドキュメントの論理的な構造を自然な形で表現できる。階層構造はドキュメントの構造に限らず様々な場面で見られ、たとえばファイルディレクトリに代表される分類もそうである。また、部品を組合せて作る構造物は階層構造と捉えることができ、これはオブジェクト指向によるデータ表現との親和性の高さに直結する。事実、XML の構造を表現するため、DOM やそれに対するオブジェクト指向言語（Java 等）による API が規定されている。扱いたい対象が持っている構造に近い形で記述できることは、システムの開発効率を高める役割を果たす。XML が表現する階層構造は、人間が直感的に把握できる方法を反映しているともいえる。

3.2.6 豊富な関連技術

XML は、現在も広く普及し続けている。この背景には、XML を利用するための強力な関連技術の存在が大きな要因の一つである。XML の関連技術は、主に W3C を中心に規格の開発が行われ、それに基づいたものがベンダーや個人によって実現される。それらの多くは安価で高度なものが多い。これらの主な物を次節で説明する。

3.3 XML 関連技術

XML には多くの関連技術が存在し、広くの場面で利用されている。以下に、主な物の説明をする。

3.3.1 XML パーサ

一般的にプログラムがファイルにアクセスする場合、そのプログラム独自の方法を用いて、ファイルの構造を理解したうえで読み書きしなければならない。XML には、XML パーサが用意されいてる。XML パーサとは、XML 文書を読み込み、その構造と情報へのアクセスを提供するソフトウェアである [36]。これによって、XML で指定されている規

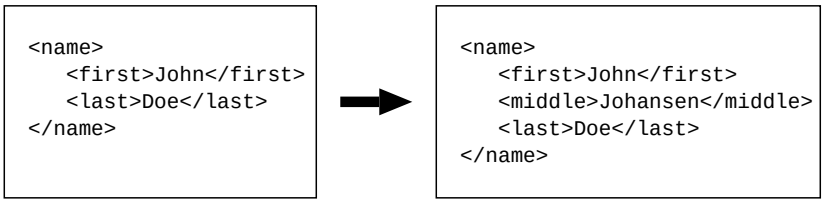

図 3.8: XML 文書の拡張

則に従いさえすれば、情報に簡単にたどり着くことができる。XML パーサは独自プログラムの中で使うことができる。つまり、アプリケーションでは XML 文書を直接参照する必要すらない。

XML パーサの他の利点として、XML 文書を拡張した場合に、アプリケーションを変更しなくても XML 文書を受け入れることができる可能性が高い点がある。例えば、図 3.8 の様に、< middle > を追加した場合でもアプリケーションのコードは変更がいらない[7]。一方、CSV 形式などでは、データに何らかの変更を加えるたびに、そのデータを利用するアプリケーション全てで設計の変更、再テストをし、導入しなおす必要がある。

XML パーサは、ベンダーや個人によって、開発され、公開されている。その多くが無料で提供されている。以下に、いくつかの XML パーサを示す。

fxp fxp[23] は、SML 用に記述された XML パーサである。1999 年に CS Dept, Univ, of Trier, Germany の Andreas Neumann 氏によって作られ、Alexandru Berlea によって管理、更新されている。fxp は XML パーサと 4 つのアプリケーションから構成されている。

JAXP JAXP[8][24] は、Sun Microsystems 社から提供されている XML パーサを含んだ Java 用のクラスライブラリである。JAXP には

- javax.xml.parsers
- javax.xml.transform

の 2 つのパッケージをもっている。J2SE1.4.0[9] 以降、この機能は標準で JDK[10] にパッケージされている。

MSXML MSXML[11][25] は、Microsoft 社が提供している XML パーサである。Microsoft 社の最初の XML パーサは Internet Explorer4 に同梱され、XML 仕様の草案が実装

[7] 要素名の変更や、要素の削除、コーディングの仕方などによっては変更が必要
[8] Java API for XML Processing
[9] Java2 Platform, Standard Edition
[10] Java Development Kit
[11] Microsoft XML Core Services

された。その後、Internet Explorer5.0 のリリースに伴い、XML1.0 の仕様が反映された XML パーサにアップグレードされた。それ以降の Internet Explorer にも付属して配布されている。ただし、MSXML は、Internet Explorer とは別にバージョンアップを続けており、Internet Explorer のバージョンではなく、MSXML のバージョンを考慮しないと開発に躓くことがある。現在の最新バージョンは、MSXML4.0SP1 である。

Xerces Xerces[26] は、Apache Software Foundation の Apache XML プロジェクトによる物である。Xerces には C++用の Xerces-C++、Jave2 用の Xerces2-Java、C++用パーサの Perl ラッパーである XML::Xerces が存在する。これらは全て無料で配布されており、ライセンスは、GNU GPL[12]に則っている。

XP XP は、Java の高機能パーサで、James Clark 氏によって開発された。James Clark 氏は、他に SGML と XML 用のパーサである SP、XSLT の Java のインプリメンテーションである XT、構文の検査が厳密である expat などを公開している。

XML::Parser XML::Parser は、James Clark 氏の expat を用いて作られた Perl ラッパーである。元は、Larry Wall によって開発されたものだが、現在は、Clark Cooper がメンテナンスしている。

上記の様に、XML パーサは数多く存在していて、プラットフォームや使用言語など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 パーサを選択する必要がある。また、上記パーサは expat と expat の Perl ラッパーである XML::Parser 以外、DTD による妥当性検証が可能な検証パーサである。

3.3.2 DOM : Document Object Model

DOM[18] とは、コードを使って XML を操作したり、プログラムのコードとやり取りするための仕様である。DOM は W3C によって 1998 年 10 月に Level1 が、2000 年 11 月に Level2 Core Specification が勧告された。また、2002 年 10 月には Level 3 Core Specification のワーキングドラフトが公開された。

DOM の概要は以下の通りである [39]。

- メモリ中に展開された XML 文書に対応している DOM オブジェクト[13](DOM ツリー)を参照、編集する言語中立な API[14]の種類と内容を規定している。

[12] General Public License
[13] XML 文書を扱うための木構造
[14] Application Programming Interface

- 規制の XML 文書を DOM オブジェクトに展開する API の内容については、処理系依存とする。
- DOM API のライブラリ、DOM API 対応の処理系においては、既成の XML 文書を DOM オブジェクトに展開する API を提供しなければならない。

DOM API の主な機能は以下の通りである [39]。

1. 既成の XML 文書ファイルを DOM オブジェクトに展開する。
2. DOM オブジェクト展開時にエラーチェックを行う。
3. DOM オブジェクトの各ノードを参照する。
4. 新規に DOM オブジェクトを作成する。
5. DOM オブジェクトを編集する。
6. 上記 4,5 に対してエラーチェックを行う。
7. DOM オブジェクトをファイルに保存する。

上記のうち W3C の DOM 仕様書で規定しているのは 3,4,5,6 項関係である。1 項は処理系依存である。しかし、DOM のアウトラインにあるように、これを提供することは必要とされている。2 項については、一般的に 1 項の API が機能を持っている。7 項も一般的に処理系およびライブラリが持っている。

DOM は、XML 文書を操作するだけの API ではなく、HTML 文書、CSS スタイルシート、その他さまざまな文書の操作を行うことができる。DOM 実装は、XML 文書や HTML 文書に特化させることもできるし、さまざまなタイプの文書で使えるように作ることもできる。DOM 仕様書にある実装 (Implement) とは、処理系あるいは、ライブラリに API を組み込むことである。実装依存とは、DOM 仕様書では規定せず処理系またはライブラリに任せるということである。[36]

DOM 仕様書にあるインタフェースは、API の Interface に対応し、Java のインタフェースではない。図 3.9 に DOM Level2 Core で用意されているインタフェースを示す。さらにコアインタフェースは、基本インタフェースと拡張インタフェースに分けられる。

基本インタフェース XML 文書以外の文書しか処理しないものも含めて、すべての DOM 実装が実装されなければならない。

拡張インタフェース XML を処理する DOM 実装のみが実装される必要がある。

DOM は標準として公開されているため、これに準拠した XML パーサが複数公開されている。上記のように 1 項、7 項の部分は XML と DOM で明文化されていないためパーサに依存しなければならないが、それ以外の部分は異なるパーサ間でコードを共有でき、開発途中でのパーサの入れ替えも容易に行うことができる。

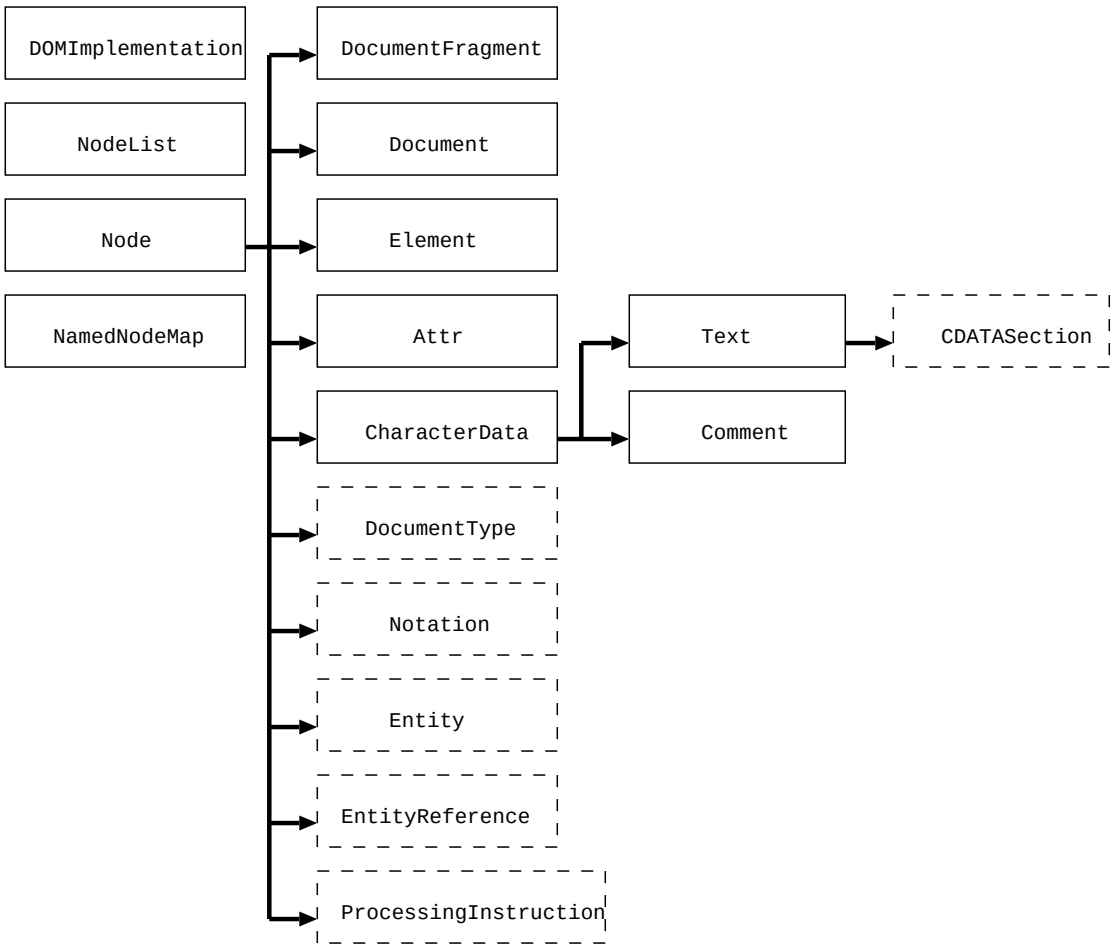

図 3.9: DOM Core Interface

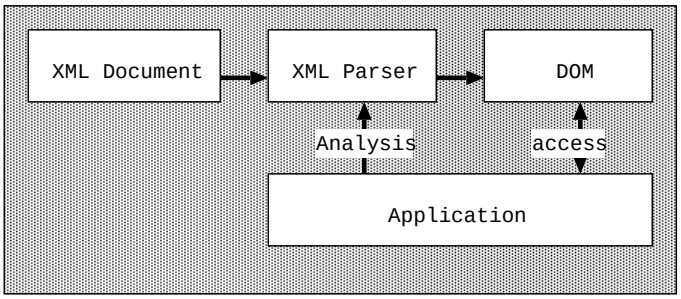

図 3.10: DOM の場合

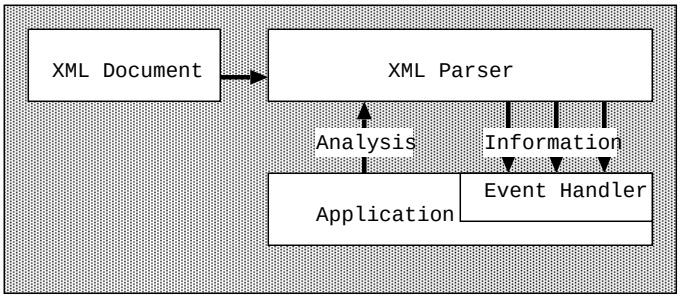

図 3.11: SAX の場合

3.3.3 SAX : Simple APIs for XML

SAX[28] とは、サイズの大きな XML 文書を効率的に解析するために開発された API である。先に示した DOM の問題として、文書を構造化する前にメモリに膨大なマップを構築しなければならないということがある。しかし、SAX は、イベント駆動型であるためメモリや時間を効率的に使うことができる。図 3.10 は DOM を、図 3.11 は SAX を表す。

SAX は、最初 David Megginson が Java のみの API として考案した。その後、Java の XML で広く採用され、事実上の標準となった。現在では、Java をはじめ C++や Perl など多くの言語で使用可能である。SAX を使用するためには、SAX 対応の XML パーサと SAX の API が必要となる。例えば、Java の場合 JDK、SAX 対応パーサ、SAX Java クラスが必要となる。ただし、J2SE1.4.0 以降には標準でパッケージされているため JDK だけで使用可能である。

3.3.4 DTD : Document Type Defintion

DTD[15] とは、Document Type Definition の略で文書型定義のことである。これは、文書の要素や属性の相互的な関係を表すものである。XML 文書には整形式の XML 文書

と妥当な XML 文書の二種類がある。整形式の XML 文書とは、正しい XML 構文を使い、全ての XML データに共通する正しくネストしたツリー構造をしていることが保障されているものを言う。妥当な XML 文書とは、整形式な XML 文書である上でさらに妥当性検証を通るものを言う。妥当性検証を行うことのできる XML パーサを検証パーサと呼ぶ。妥当性検証とは、XML の構文規則以外に決めた独自規則にも一致していることを保障するということである。この XML 文書の構築方法を定義する一連の独自規則をどのように指定したらよいかを DTD で記述する。

ほとんどの XML アプリケーションには以下の要件が必要である。[36]

- 文書構造をできるだけ厳密かつ一貫した方法で実現する。
- 文書構造を他のアプリケーションや人に伝える。
- 必須の要素が存在しているかチェックする。無い場合はそれらを含める様に作成者に注意を促す。
- 許可されて無い要素が含まれていないかチェックする。また、作成者がそれらの要素を使えないようにする。
- 要素の内容、ツリー構造、要素の属性を強制する。
- 特に指定されていない属性値に規定値を提供する。
- 標準の文書形式とデータ構造を使う。

しかし、これらをアプリケーションで維持するのは困難である。これはほとんどの XML アプリケーションに共通の問題であり、したがってより標準化されたアプローチである DTD が必要である。

DTD の最大の特徴は、DTD と検証規則が XML1.0 で体系化されているという点である。XML の急速な普及とともに汎用的にデータを記述するうえでの DTD の制約が明白になった。以下に主な制約事項を表す [36]。

DTD は拡張可能でない DTD を使って XML ボキャブラリの規則を記述するときは、それらの規則を全て単一の DTD に記述しなければならない。スライシングで実行系列を表す場合に、実行系列、実行時点、変数など　別々のボキャブラリを作ることができない。全て単一の規則とし、その DTD を変更しなければならない。

各文書に一つの DTD XML 文書 1 つに対して DTD は 1 つである。これにより 1 つの XML 文書を複数の記述に対して検証することができない。実行系列を XML 文書にした場合、XML 文書内に検証用の DTD が指定してある。他の DTD での検証がしたい場合、XML 文書に変更が必要になる。

名前空間のサポートが不十分である XML 名前空間には次の 2 つの利点がある

- 要素型の名前の競合を回避できる
- 同じものを意味する異なる名前文字列を解決できる

名前空間が使われている場合は、各名前空間の要素型を全て DTD に含めなければならない。これはさまざまなソースから複数の名前空間を使う目的に反している。スライシングで、複数のプログラムから、機能抽出した場合、これらの複数のスライスを統合すると、同じタグが同一視される。どのスライスに含まれていたものか判別したいとき、名前空間は重要な役割を果たす。

データ型が不十分 DTD が実際に処理できるのはテキスト文字列のデータ型だけである。一部の文字列 (ID、NMTOKEN など) にはいくつかの制限を適用することも可能であるが、明らかにこれらは不十分である。数字のデータ型は存在せず、日付、時間、URL などの複雑な構造を処理することはできない。実行系列を表現する場合でも行番号や、アドレス値など、必ず数字データが入ることが前提であっても、DTD では規定できない。これを実現するにはアプリケーションで文字列として検証しなければならない。

継承が無い 近代的なオブジェクト指向システムは、新しいオブジェクトを既存のオブジェクトの表現を使って確実かつ有効に記述するという考え方に基づいている。要素型がオブジェクトクラスに相当するものだとすると、1 つの要素型を他のもので記述することができればきわめて有用である。しかし、DTD ではそれができない。

DTD のオーバーライド 外部 DTD が文書インスタンス内の内部 DTD のサブセットによって完全に無視またはオーバーライドされるということは、DTD の規則に、それが関連付けられている文書がしたがっている保証が無いということである。

DOM は DTD をサポートしていない DOM は XML 文書を扱うとき良く使われる技術である。しかし、DOM は EBNF を処理したり、DTD の文書モデルにアクセスしたりすることができない。

3.3.5 XSLT : XSL Transformation

XML のスタイルシート記述言語である XSL[19] には、変換機能とフォーマッティングオブジェクト[15]機能が含まれていた。この中で、変換機能の部分は単体でも非常に有用で、XSL 以外の目的でも役に立つことから、分離し単独で XSLT[20] として W3C の勧告となっている。よって、XSLT は、XSL の一部として使われることもあれば、フォーマッティングオブジェクトに依存することなく XSLT 単独で用いることも可能である。

XSLT とは、XSL Transformation という言葉のとおり、変換のための仕様である。XSLT は、図 3.12 のように、任意の XML 文書を読み込み、XML 文書、HTML 文書などに変換す

[15] Formatting Object

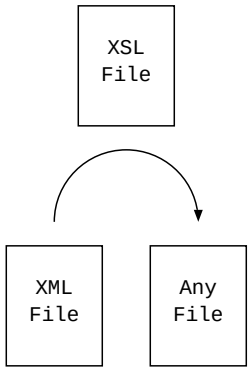

図 3.12: XSLT の働き

るために用いられる。そして、この変換時に使用されるのが XSL である。XSLT は、XSL で記述されたテキストファイル (XSL ファイル) を触媒とし、XML 文書を変換する [37]。

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「この入力の場合に、以下の出力を生成する」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はなく、複数の規則が同じ入力に一致する場合は、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読み取りされない。

3.3.6 XPath : XML Path Language

XSLT コードでは、<xsl:template match=”hoge”> や、<xsl:value-of select=”foo” /> などのように、XML 文書中の要素ノードなどをパターン指定していた。この方法は、XML 文書の階層構造が簡潔で浅い場合は容易に実現できる。しかし、XML 文書の階層構造が複雑で深い場合は、XML 文書の特定の部分を処理する方が効率的 [37] である。XML の関連技術には、このような処理をする XPath が存在する。

XPath[21] は、XSLT と XPointer[22] との間で共用している機能について、共通の文法と意味論を用意しようという努力の結果である。XPath の主な目的は、XML 文書の一部をアドレスすることである。この目的のために、XPaht は文字列、数値、ブール値を操作するための基本的な機能を提供する。XPath は、URI や XML 属性値の中で、XPath を使いやすくするため、コンパクトな非 XML 文法を用いる。XPath の名前は、URL のように、XML 文書の階層構造を通じてナビーゲートするためにパス表記を使用することから名づけられた。XPath は、アドレッシングの利用に加えて、照合 (あるノードがあるパターンに合致しているかどうかのテスト) に利用できる自然なサブセットを持つ。XPath は、1999 年に勧告として 1.0 が公開されている。

第4章　XMLを用いた動的スライサの提案と実現

4.1　XMLを用いた動的スライサの提案

動的スライサの開発コストは非常に高い。これは以下の要因によるものと考える。

- 実行系列のデータサイズが大きくなる
  実行系列のサイズは、実行された命令の述べ数に比例して大きくなる。
- 実行環境が必要
  実行系列を抽出するためには、プログラムを実行するための実行環境が必要である。
- データの表現の難しさ
  静的ツール用では、AST + 記号表 + 型情報 + クロス参照 と、よく知られたデータ構造が存在するが、動的用の実行系列は、コンパイラの教科書や論文にもほとんど記述が無く、モデル化や表現の経験の蓄積が少ないため、データスキーマを開発者が決めなければならない。

しかし、動的スライシングはソフトウェア工学的に大変有用であり、その実装である動的スライサも、また、有用であると考える。ところが、上記の理由から動的スライシングの研究は現在も活発に行われているにもかかわらず、小さな概念的な言語に対するものばかりで、フルセットの言語に対応し、一般に公開された動的スライサは我々の知る限り存在していない [10]。そこで我々は、実行系列を中間データとして XML で表現することによって動的スライサの開発コスト削減の可能性を探る。

XML 文書を中間形式として導入するには、DTD によるデータスキーマが必要だと考える。プログラムの実行系列はさまざまな形式で表現可能であるため、実行系列の抽出を行う実行環境の開発者と、依存解析を行うツールの開発者でデータスキーマについてのディスカッションが必要となる。ディスカッションを行わない場合、両者の意図が食い違い、ツール開発の条件に満たないデータスキーマを抽出してしまう可能性が高い。また、データスキーマを汎用的な DTD で定義することにより、複数のツール開発者でデータスキーマに基づいた開発ができ、データの共通性が生まれると考える。

我々は、動的スライサのターゲット言語として、ANSI C プログラムに着目した。4.3.1 節で説明するように、ANSI C はその構造上、動的スライスの適用性が高いと考える。我々

はANSI Cのフルセットに対応した動的ACMLをゴールだと考えている。しかし、ANSI Cは非常に大きな仕様である。よって、この研究の初期段階として以下の2つのポイントを考慮し、ANSI Cのサブセットに対し、データスキーマを提案した。

- スカラー変数のみのANSI Cを対象とした動的スライサ
- ポインタ変数を含んだANSI Cプログラムを対象とした動的スライサ

前者は、動的スライサの有用性と、XMLが動的スライサに利用できるかを確認するためで、後者は、より実用的なANSI Cプログラムに対する動的スライサにはポインタが必要不可欠だという考えである。

4.2 XML導入の利点

我々は、XMLが動的スライサの開発コストの削減に影響を与えると目論む。以下に利点となりうるXMLの特徴を挙げる。

4.2.1 テキスト形式

XML文書はテキスト形式で保存されている。これは、ファイルを見ることによって中間データを人間の目で確認することが可能なことを指す。その上でXML文書は構造化文書のためXMLパーサを通すことによりアプリケーションからも手軽に扱うことでき、XML関連技術を利用することにより、同じテキスト形式のCSV形式などではアプリケーションが構文解析、字句解析をしなければならないが、その分のコストが分離できる。

動的スライサに限らず中間データを持ったアプリケーションは、中間データの形式にバイナリ形式が使われていることが多い。これは、サイズや実行速度などの実行効率を考慮した結果である。しかし、バイナリ形式は開発者以外には理解が困難であり、バイナリ形式の情報を中間データとして扱う場合、明らかに開発コストの増加につながる。テキスト形式であれば、認知が容易であり、中間データの理解に要する労力を削減することが可能であると考える。

4.2.2 自己記述性

XMLは、構造化文書である上に新たに言語を作ることができるメタ言語である。SGMLでは、ラベルの付いたタグにより情報構造が表現される。HTMLではタグの種類が固定され、Webブラウザの標準サポートも規定のタグに限られる。このため、要求される構造に合わせた情報の表示や入力を、効果的、効率的に行うことができない。しかし、タグの種類を自由に定義することができるメタ言語のXMLは、タグの種類と利用方法を規定することにより柔軟に要求される構造に対応することができる。

動的スライシングで扱う実行系列は、モデル化や表現に経験の蓄積が少ない。つまり、開発者自身が自己の判断で必要と考える情報を取得する。同一の開発者のみが扱うデータスキーマであれば問題は少ないが、複数の開発者が単一のデータスキーマを扱うことを前提とする場合、データスキーマのドキュメントが必要になる。しかし、XML の自己記述性を用いれば、取得した情報の説明を XML 文書に埋め込むことができる [36]。

4.2.3 柔軟なデータ表現

XML では、基本的に木構造を表すことが容易である。木構造は、多くのデータ形式を表すのに十分強力で、概念が簡単なので扱いやすい。。例えば、プログラムの抽象構文木なども木構造である。また、XML にはリンクがあるため、グラフ構造や、表構造、リスト構造なども容易に表すことができる。

動的スライシングで必要な実行系列は基本的に表構造をとると考えている。しかし、依存関係やジャンプなどを表すリンク構造、変数に関する情報を収納する表構造、構文情報を表す木構造など、単純な表構造では表現できない構造も容易に表現できると考える。

4.2.4 単純な構造

XML 文書は基本的にタグの組み合わせの木構造である。タグ自体は < と > にはさまれた簡単な形式であり、非常に理解しやすい。実行系列を XML 文書化したとき、S 式や CSV 形式など、他のテキスト形式より複雑になっては、理解に要する時間が開発コストに影響する。単純な構造は理解のしやすさであり、開発効率の妨げにならないということである。また、単純な構造は習得の容易さも持ち合わせている。

単純な構造は、手作業での作成も可能とする。XML 文書を手作業で作成することは難しくない。これによりテストケースを特別な実行環境なしで作成することが可能であり、実行環境と依存開発ツールを平行して開発することが可能であることを表している。

4.2.5 再利用性

再利用性にはいくつかの側面があると考えている。以下にそれらを示す。

- 複数のスライシング基準での利用
  我々の考える動的スライサは、最初に入力値を与え、実行環境で実行することにより、実行系列を XML 文書で抽出する。この時点で、スライシング基準である (行番号、変数名) は決定していない。つまり、入力値を固定した状態ではあるが変数や行願望の指定を変化させて同じ実行系列に何度もスライシングをすることが可能である。これは、毎回、実行系列を抽出する方式の動的スライサより、実行系列抽出の回数が減り、複数回実行時の実行コストは上がると考えられる。

- 複数の動的スライサでの利用
  動的スライシングの様々なアルゴリズムで要求する情報を動的 ACML が保持している場合、その XML 文書は、複数の動的スライサで利用可能である。これは、特定のプログラムに様々な動的スライシングを行えるということであり、特定の動的スライサでは不得意な部分を他の動的スライサでカバーすることも可能である。また、開発段階において、同一のデータスキーマに対して開発を行うことは明らかに開発コストを削減させられるということだ。しかし、これを行うためには、データスキーマが様々なアルゴリズムで必要とする情報を持っていることが前提であり、様々なアルゴリズムに対応させたデータスキーマは大きく複雑で、冗長なものになってしまう可能性も持っている。

- 複数の動的 CASE ツールでの利用
  DTD で規定したデータスキーマは動的スライシングのみならず他の動的解析に使える可能性を秘めている。これにより、プロファイラやプログラムシミュレータなど動的 CASE ツールのプラットフォームとして利用できる可能性もある。しかし、これも上記項目と同じように、様々なアルゴリズムを適用する場合には、非常に多くの情報を保持しなければならず、情報が増えることによってデータスキーマが複雑になり、理解が困難になり、逆に開発コストがかかってしまう可能性も持っている。

4.3 実現した動的 ACML

XML が実用的に活用できるか確認するためには、フルセットの言語をサポートするべきだと考える。しかし、フルセットで書かれたプログラムの実行系列は抽象度、変数の種類など多種多様である、これらを全てをマークアップすることは、大変困難であり、また冗長である。そこで、本研究の初期段階として ANSI C のサブセット (5.2 節に詳細) に対し、特定の動的スライサの要求を満たすものをマークアップすることとした。

実現した動的スライサの構成要素は以下の 3 つである。

- 動的 ACML
  ANSI C 用に DTD で定義したデータスキーマ

- XCI
  ANSI C プログラムを実行し、実行系列を XML でタグ付けされた XML 文書として出力するインタプリタ

- 依存解析ツール
  動的 ACML を基に、動的スライスを抽出する依存解析ツール

図 4.1 に動的スライサの構成を示す。まず、XCI に、ANSI C プログラムと入力値を与え、タグ付けされた実行系列 (DACML 文書) を取り出し、その DACML 文書に対し、

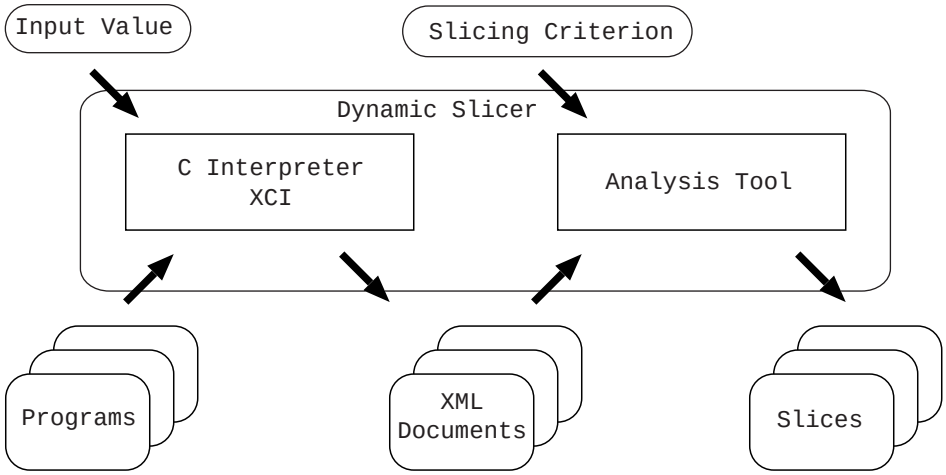

図 4.1: 動的スライサの構成

XML パーザを通じて DOM や他の関連技術を用い、動的スライサを抽出する。

4.3.1 ANSI C を対象にする理由

本研究において、我々は ANSI C のみをターゲットとする。その理由は、ターゲットを一つに絞ることで、我々のゴールに向かう第一歩と考えたからである。では、「なぜ ANSI C を選択したか」の主な理由を以下にあげる。

- 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。
- 特定のコンパイラ (例えば GCC の独自拡張言語) を対象にすると、独自拡張した文法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。
- 一般に C 言語は、その文法構造上バグの要因 (ポインタによるエイリアス、キャスト、副作用、NULL ポインタなど) が多く、動的スライサの利用が多いと考える。

以下に、上記 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、その他ソフトウェアは、ほとんど C 言語で書かれていて、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 言語は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 言語を各々拡張する方針を

取った。その結果として仕様の異なる C 言語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となった。

そこで、アメリカ国内標準協会（ANSI[1]）は、1983 年、C 言語に関する現代的で統括的な定義を行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 言語に比べ、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となった。

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いくつか挙げると、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 の標準規則には無い機能で、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 の文法を複雑にする要因となっている。

ANSI C の性質

ANSI C の目的は、さまざまに拡張された C 言語を、一つの標準規則にまとめることにあった。その方針として「C の精神を守る」と、委員会の協議の場で何度も言われた。この「C の精神」を書き下すのは難しい。解説 (Rationale) では、以下の方針を「C の精神」としてあげている [40]。

- 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もたらした。しかし、その反面で、その言語構造の単純が、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

[1] American National Standards Institute

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った。
以下に、代表的な ANSI C でのバグを例示する。

- 副作用
  ANSI C では、副作用の起きる順序を決めていない。ある変数に対して副作用が生じる式がある場合、同じ式の他の箇所でその変数を参照してはいけない。なぜならば、副作用がいつ起きるかは、未定義[2]だからである。　例えば、図 4.2 の場合、i＋＋という式を実行すると i の値が変わる。この振る舞いを副作用という。i は i＋＋の他に、配列 a の要素を指定するために参照されている。これは、一つの式の中で副作用のある式で参照された i を配列のインデックスとして参照しているので未定義である。また、図 4.3 の例では、∗ の評価順序は不定[3]であるが、i＋＋はどちらから実行されても同じ値になるように思える。しかし、i＋＋の副作用は、i＋＋∗i＋＋という式が終了するまでのいつか、ということしか保障されていない。ここでは、一つの式で、副作用が発生する操作を二回行っている。これは未定義であるため、プログラム処理系はどのような結果を返してもよい。

```
int a[10];
int i = 0;

while (i < 10)
   a[i] = i++;
```

図 4.2: 副作用を起こす式で参照された変数の二重参照

```
int i = 7;
printf("%d\n", i++ * i++);
```

図 4.3: 副作用を持った式が二つある式

- 評価順序
  ANSI C では、式の評価順序は不定である。評価順序が不定ということは、図 4.4 のようなコードで、関数 foo() と bar() で、どちらが先に呼ばれるか決まっていないことを表す。このコードでは、両方の関数内で大域変数を利用している。関数の呼び出し順序が決まっていないので、場合によって予期しない値を示す可能性がある。ただし、例外として”&&”、”||”、三項演算子、カンマ演算子は評価順序が決まっている。このため図 4.5 のようなコードが存在した場合、意図した結果と異なる場合がある。このコードで *if* 文の条件部分で (a == b&&c = d) とあるが、&& は、評

[2]JIS(Japanese Standards Association：日本規格協会) では未定義の動作は「可搬性がない若しくは不正なプログラム構成要素の使用における動作、又は不正なデータ若しくは不確定な値を持つオブジェクトの使用における動作であり、この規格が何ら要求を課さない動作。未定義の動作に対して、その状況を無視して予測不可能な結果を返してもよい。」と定義してある。

[3]JIS C では、不定のことを「未規定の動作」という表現を使っており、定義は「正しいプログラム構成要素及び正しいデータに対する動作であり、この規格が明示的に何ら要求を課さない動作。」となっている。

価順序が左から右であるため、a == b が偽であると、その時点で c = d は評価されない。

```
int G = 0;
main(){
   int a;

   a = foo() + bar();
}

int foo(){
   G++;
   return G;
}

int bar(){
   G = 2;
   return G;
}
```

図 4.4: 評価順序の不定

```
int a, b, c = 0, d = 1;

if(a == b && c = d){
   d = 2;
}
printf("%d %d", c, d);
```

図 4.5: 評価順序の決まった演算子

- ポインタ

  ANSI C ではポインタ演算を利用できる。C 言語のポインタは、Java などと違って、非常に自由に記述することが可能である。このため、バグの要因になりやすい。例えば、NULL ポインタへの代入が考えられる。NULL ポインタとはどこも指していないポインタであることを保障している。どこも指していないポインタに値を代入ができないことは明らかだ。しかし、コード上で明示的に NULL を代入している場合以外、ポインタが実際どこを指すのかプログラムを実行しなければわからない。これはプログラムが大きく複雑な状態では原因を探すのに非常に困難なバグの一つである。
  また、ポインタの初期化も問題になることが多い。ポインタはどこか正しい実態を指す必要性があるにもかかわらず、C ではそれを保障しない。ポインタは宣言された段階ではどこを指しているか不明である。初期化されていないポインタは、他で利用している領域を指すかもしれないし、アクセスが禁止された領域を指すかもしれない。初期化していないポインタの使用は初期的なミスであるが、実行ごとに違う動きをする可能性があるので、非常に見つけにくくなることもある。

- 定義されていないメモリ空間へのアクセス

  ANSI C は、ポインタ演算などを利用し、メモリ空間への自由なアクセスが可能

である。しかし、この自由さはバグの要因にもなる。例えば、配列 a[10] があったとき、a[15] には何があるかわからない。また、malloc で領域を確保し、free した領域を利用しようとした場合も、その空間は、不正にアクセスできてしまう。

- エイリアス
  C ではポインタを使うことによってエイリアス (別名) を作ることができる。例えば、図 4.6 で、∗a と ∗b は、c の別名である。これにより、(∗b) + + で c の値が増加している。このエイリアスは非常に解析が困難である [41]。例えば、図 4.7 のようなコードの場合 x の使用に到達する定義が最初の x であるのは ∗p が x のエイリアスでないときである。

```
int *a, *b, c;

c = 1;
a = &c;
b = a;
(*b)++;
printf("%d", c);
```

図 4.6: エイリアス (別名)

```
x = ...
*p = ....
y = x;
```

図 4.7: エイリアス解析が必要なコード

これらのように、ANSI C は、プログラム言語として大変有用であるにもかかわらず、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 SML) 以上に、テストや保守に多大なコストおかけざるを得ない。また、デバッガ、静的スライサ、プロファイラ、メモリリークチェッカなどは、実行系列に基づいて再帰的な依存解析を行わないため、実行系列を用い、プログラム内の依存関係を解析する動的スライサによるサポートは、必要なものと考える。

4.3.2 実行系列の情報

動的 ACML とは、我々が提案する ANSI C プログラムの実行系列用マークアップ言語である。動的 ACML は、XML の DTD を用いたデータスキーマであり、動的スライサの中間言語として作用する。この節では、この DTD の設計概念を示し、実際、動的 ACML がもつべき情報を議論する。

実行系列のための DTD 設計

ANSI C プログラムの実行系列をマークアップする場合、そのモデル化が必要となる。実行系列は、構文情報と違い、規定があるわけではない。これにより、どのような構造

で、どのような情報を持つかは設計者の判断になる。また、動的スライサに必要な情報の粒度や、スライスを抽出する際、もっとも適当な抽象度は何か、などを決めなくてはならない。つまり、DTD 設計段階での大きな問題点は以下の 3 つである。

- どんな情報を取ることが妥当か
  ANSI C の実行時の情報には多種多様なものが存在する。例えば、構文情報、実行順序、評価順序、型情報、シンボル名、ポインタアドレス、変数値、変数のサイズ、変数の種類などこれらはその一例である。4.3.3 節で説明するが、動的スライサの実行系列はサイズが大きくなりやすい。よって、情報を全て取得保持しておくことは非常に困難なので、動的スライサに必要な情報を取捨択一しなければならない。

- どんな粒度で表現するか
  ANSI C の動的情報の取得は様々な粒度で行うことができる。ファイルや関数の実行単位で情報を取得すれば粗粒度で表せる。また、変数の値やサイズ、ポインタアドレスなどの要素で取得すれば細粒度で表せる。動的スライシングは、変数の使用と定義を基にステートメント間のデータ依存関係と、制御命令の制御範囲を基に制御依存関係を解析し、動的スライスを抽出する技術である。これより、変数などの細粒度な情報が必要なことがわかる。しかし、x = y + z の'+' のような構文情報は動的スライシングに必要とはしない。よって、全ての情報を細粒度で取得するのではなく、細粒度で取得しなければならない部分、粗粒度で十分な部分が混在すると考える。

- どんな抽象度で情報を保持するか
  実行系列の表現方法には、様々な抽象度が存在する。抽象度の最も低い状態はヒープやスタック、スタィックのメモリ空間の履歴をマッピングすることである。また、違う抽象度であれば、変数のシンボル名、変数値など構文に近い形でも取得することができる。しかし、これらにはそれぞれ問題がある。メモリ空間のマッピングは全てのデータ取得が可能であるが、プログラムの実行時に使用されなかったメモリ空間など必要のない情報が多数存在し、また、依存解析時に必要な抽象度まで引き上げなければならず開発コストが増える。逆に、全て高い抽象度で取得した場合、例えば変数のシンボル名から、その変数のメモリアドレスは判別できない。

動的スライサに必要な情報

実行系列は、主に変数の情報に基づいて依存を解析するため、実行時点ごとの変数についての表になる。以下に図 4.8 のプログラムに対して "`i = 1, a = 1, b = 0, c = 0, x = 1, y = 2`" を入力値としたときの実行系列の一例を図 4.9 に示す。しかし、この

```
10 : while(i < a){
11 :     b += x + y;
12 :     i++;
      }
13 : printf("%d", b);
```

図 4.8: サンプルプログラム 4

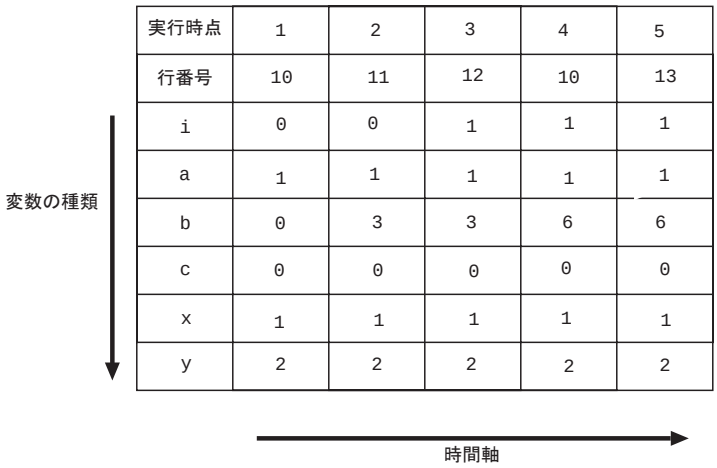

図 4.9: 実行系列の一例

表は、使用していない変数に対する情報が冗長であり、また、制御文などの情報を持っていないため動的スライサに適したデータスキーマとは言いがたい。

動的スライシングは、実行時点の変数の使用と定義を基に、実行時点間の依存関係を求める技術である。このため実行時点を識別するインデックスと変数の使用、定義の区別が必要となる。さらに実行時点は、構文情報の一行に相当する。このため、行番号を保持すれば、任意の実行系列から、基となった構文情報を取得できる。
また、データ依存を解析するためのキーとなる変数については、静的な情報としてシンボル名、型情報などがある。また動的な情報として、変数の値、アドレス値、サイズなどが考えられる。
さらに、制御依存関係を解析するために、制御文の制御範囲が必要となる。制御範囲とは、分岐文およびループ文が実行された実行時点 q に対して、実行時点 q の直後に引き続いて、その分岐文およびループ文内で実行時点の集合のことである [35]。また、制御文が実行されたときの条件命令の真偽値も動的に決まる。これらを踏まえ、以下に最低限必要な情報をまとめる。

- 行番号
- 実行時点を識別するインデックス
- 制御文の制御範囲
- 制御文の条件命令の真偽値
- 変数の使用、定義の別
- 変数のシンボル名
- 変数の型情報
- 変数のアドレス値
- 変数のサイズ
- 変数の値

4.3.3 動的 ACML のコンセプト

4.3.2 節で議論した情報より、動的スライサに適したデータスキーマを考える。動的 ACML のコンセプトを以下に表す。

- 情報取得の粒度
  我々は、タグ付けの対象を関数やステートメントではなく、変数のシンボル名や変数値などとし、動的 ACML の粒度を細粒度とした。これは、動的スライシングが、細粒度の情報を必要とする技術だからである。しかし、実行系列を細粒度で取得することは非常に情報量が大きくなる。このため、動的スライシングで必要とならない構文情報 (例えば、四則演算子など) を極力削除することで、情報量を減らした。

- 抽象度の設定
  実行系列には、多種多様な抽象度が存在する。例えば、ヒープ領域、スタック領域、スタィックなメモリ空間を表せば、非常に低い抽象度で情報が取得できるし、変数のシンボル名や値を実行時点ごとに表にすると、非常に高い抽象度で情報が取得できる。基本的に動的スライシングは、高い抽象度の情報を利用する。しかし、高くすると、アドレス値などの情報は隠蔽され利用できない。また、ヒープ領域やスタック領域のマッピングといった低い抽象度でのデータの取得は、スライスの解析時にデータの加工 (例えば、その実行時点で利用されたメモリ空間から、変数のシンボル名やアドレス値を取得する作業) が必要になり、アプリケーションの開発コスト

が増えると考える。これらを踏まえ、我々は、基本は高い抽象度で、必要な部分 (例えば変数の値、シンボル名、アドレス値など) だけ低い抽象度として情報をもたせることにした。

- データの冗長性の排除
  プログラムの実行時にはさまざまな種類の情報が存在する。プログラムの構文情報や変数のシンボル名、値、メモリのアドレス空間などである。また、実行系列はその性質上、サイズが大きくなりやすい。例えば、10 個の命令でできたプログラムで 5 個の命令が while 文の制御に依存している場合、while 文が一回実行だけ実行されれば 10 個の命令が実行されたことになるが、while 文が 100 回実行されると、500 個以上の命令が実行されたことになる。さらに、一般的に XML 文書はタグ付けを行うため元の情報よりサイズが大きくなる。これらの理由から、データの冗長性を省くことによって、XML 文書のサイズを増やさないようにする。例えば、各実行時点でヒープ領域、スタック領域をマッピングすると、ある実行時点 p で使用されていない変数 x の情報も含まれる。これは明らかに冗長である。こういった場合は、各変数について、差分を持つ方が冗長性が減らせる。

- 構文情報の排除
  動的スライシングは、変数の使用と定義からなるデータ依存関係、制御命令の実行に依存する命令を表す制御依存関係からなる。これらの依存関係を解析するためには一部の構文情報しか必要としない。例えば、`z = x + y;` と `z = F(x, y);`[4] いう構文情報があったとき、`F` に副作用が無ければ、動的スライシングに必要な情報は一緒である。このように、解析に必要の無い構文情報が存在する。これらを排除することによってデータスキーマの構造を簡単にし、理解の容易性を高め、XML 文書のサイズを減らす。また、静的な構文情報用のデータスキーマとして ACML[31] が存在するため、それを用いれば、省いた構文情報を利用することも可能である。

- 動的スライシングに必要なデータ依存、制御依存等の解析に必要な情報を持つ
  上記二項目のように、データスキーマは、理解度とサイズの問題のためできる限り不必要な情報を削除したい。しかし、データ依存、制御依存などの解析に必要な情報を保持していなければ、実行系列として利用できない。これにより、依存関係抽出のために必要な情報を見極め、それらをデータスキーマに組み込まなければならない。

[4] `F` は関数名

- よく使う情報は冗長であっても必要である。
  実行系列はサイズが大きくなりがちであるため冗長性をできる限り省きたいというのが、上記で述べたコンセプトの一つである。しかし、たとえ冗長であってもよく使う情報は持たせるべきだと考える。例えば、制御命令と制御範囲は、制御依存を求めるためのキーポイントである。この情報は、開発者にとってわかりやすくあるべきである。そこで、制御範囲の命令を子要素として持たせる。この構造を XML 文書とすると制御範囲を開発者が目で確認することができる。しかし、これをアプリケーションで扱う場合、子要素から親要素へのアクセスは、親要素から子要素へのアクセスより手順が多いことがあるため、一概によいとは言えない。そこで、子要素から親要素へリンクを持たせると、一手順で親要素へアクセスが可能である。これは、制御範囲という同じ情報を表す冗長なデータであるが、開発者のデータ理解と、アプリケーションからの操作のしやすさは両方必要である。

- 実行系列の抽出側と解析側の使いやすさの相違
  実行系列の抽出側と解析側の両者の意図は、ずれが生じやすく、両者にとって都合の良いデータスキーマ定義を考えるのは容易ではない。例えば、抽出側は、式の評価順序に素直にデータを抽出できると楽である。評価順序はコンパイラ依存であるため、図 4.10 のような抽象構文木の場合、右再帰で、t, =, s, +, y, =, x と評価するのが自然である。つまりこれは、変数 t が参照され、変数 s が定義されて、変数 y が参照されその結果を演算し、変数 x に定義したということになる。しかし、解析側は、使用と定義の関係を簡単に参照したい。図 4.11 のように、使用された変数、定義された変数がデータスキーマで固まっていた方が明らかに扱いやすい。他にもよく使う情報は解析のときに取得しやすい形になっている方が、解析側としては楽である。

```
10 : x = y + (s = t);
```

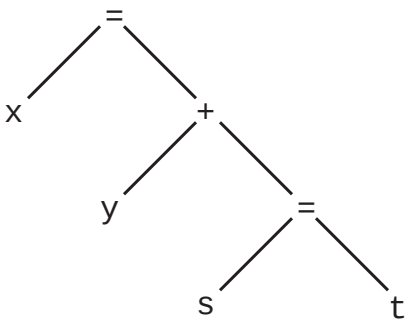

図 4.10: サンプル式と評価順序

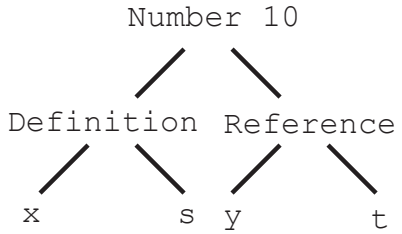

図 4.11: 解析側の扱いやすい例

4.3.4 動的 ACML：Dynamic ANSI C Markup Language

我々は、4.3.2 節での議論を基に、2 種類の動的 ACML を定義した。1 つは Korel と Laski のアルゴリズムを使った動的スライサに対するもので、動的スライシングの最低限の機能を実現している。もう 1 つは、前のアルゴリズムと Agrawal のアルゴリズムの両方に使えるデータスキーマを考えた。Agrawal のアルゴリズムは、前者よりもっと実用的で、ポインタを考慮した動的スライシングが行える。

定義した動的 ACML は、付録 A と付録 B に示す。動的 ACML で取得した情報を以下に示す。

- Korel と Laski のアルゴリズムの動的 ACML(付録 A 参照)
  - `hist_root`
    XML 文書のルート要素を表す。
  - `line`
    実行された式文に対応する。属性として実行時点 (`id`) と行番号 (`number`) を持つ。子要素として、命令の種類 (`loop`、`call`、`branch`、`assign` のいずれか) を持つ。
  - `loop`
    ループ命令に対応する。子要素として、条件命令で使用された変数の集合、条件命令の真偽値、制御範囲を持つ。属性として、分岐命令の種類を持つ。
  - `branch`
    分岐命令に対応する。子要素として、条件命令で使用された変数の集合、条件命令の真偽値、制御範囲を持つ。属性として、分岐命令の種類を持つ。
  - `call`
    関数呼び出しに対応する。関数呼び出しはライブラリ関数の一部を特別に使った。これらを区別するために用意した。子要素として、定義された変数の集合、使用された変数の集合を持つ。属性として、呼び出した関数名を持つ。
  - `assign`
    代入命令に対応する。定義された変数の集合と、使用された変数の集合を持つ。
  - `block`
    制御部命令の制御範囲を表す。親要素に存在する制御命令に制御依存している命令を子要素として持つ。
  - `def`
    定義を表す。子要素には変数の集合を持つ。
  - `ref`
    使用を表す。子要素には変数の集合を持つ。

- variable
  変数を表す。子要素として、変数名、型情報、変数値を持つ。
- type
  変数の型情報を持つ。
- name
  変数のシンボル名を持つ。
- value
  変数の値を持つ。
- truth
  制御文の真偽値を持つ。

- Agrawal のアルゴリズムに対応した動的 ACML(付録 B 参照)
  - root
    XML 文書のルート要素を表す。
  - line
    一つの式文に対応する。子要素として、selection、iteration、expresstion のいずれか一つを持つ。また、属性として、実行時点 (id)、行番号 (number) を持つ。また、制御依存 (depend) を属性のオプショナルで持つ。
  - iteration
    ループ命令に対応する。子要素として、条件命令 conditional と、制御範囲内で実行された式文 line の集合を持つ。属性として、ループ命令の種類と条件命令の真偽値を持つ。
  - selection
    分岐命令に対応する。子要素として、条件命令 conditional と、制御範囲内で実行された式文 line の集合を持つ。属性として、分岐命令の種類と条件命令の真偽値を持つ。
  - expression
    expression には代入命令と条件命令がある。代入命令は一つ以上の定義した変数を持つ。条件命令には定義した変数が存在せず、使用した変数が一つ以上存在する。
  - conditional
    条件命令を表す。条件命令には一つ以上の使用した変数が存在し、定義した変数は存在しない。
  - assignment
    代入命令を表す。代入命令には一つ以上の定義した変数が存在し、0 個以上の使用した変数が存在する。

- `variable`
  スカラー変数を表す。子要素は、シンボル名、ポインタ値、サイズ、値である。
- `struct`
  構造体を表す。子要素は、シンボル名、、オフセット値、ポインタ値、サイズ、値である。オフセット値には構造体のメンバ名が入る。
- `array`
  配列を表す。子要素は、シンボル名、、オフセット値、ポインタ値、サイズ、値である。オフセット値には配列の要素を表すインデックスが入る。
- `name`
  変数のシンボル名を表す。
- `offset`
  オフセット値。構造体の場合はメンバ名、配列の場合はインデックスを持つ。
- `address`
  ポインタアドレスを持つ。
- `size`
  変数のサイズを持つ。
- `value`
  変数の値を持つ。

上記のように、動的 ACML を定義した。以下に、動的 ACML の適用例を示す。

- 変数
  変数の取得情報は、動的スライシングの中で最も重要である。Korel と Laski のアルゴリズムはスカラー変数のみなので、`variable` のみとしたが、Agrawal のアルゴリズムはポインタ解析を行うので、構造体、配列、スカラー変数を区別した。以下に Korel と Laski での変数 (図 4.12)、Agrawal での変数 (図 4.13)、Agrawal での配列 (図 4.14)、Agrawal での変数 (図 4.15) をそれぞれタグ付けし、例を示す。値は全て 1 とする。

- 制御命令
  制御命令は、データ構造と共に動的スライシングの依存関係を求めるのに重要な制御依存関係に起因する命令である。Korel と Laski のアルゴリズムと Agrawal のアルゴリズムでは基本的に同じ扱いであるが、Agrawal のアルゴリズムではより扱いやすいようにリンクを用いて制御命令を直接指定したり、条件命令の真偽値を属性とし改良した。例として、図 4.16 のコードを実行したときの Korel と Laski のアルゴリズム用 (図 4.17) と Agrawal のアルゴリズム用 (図 4.18) を示す。ただし、`x = 1` とし、while 文の実行時点は `a7` とする。

```
int a;

<variable>
   <type>int</type>
   <name>a</name>
   <value>1</value>
</variable>
```

図 4.12: 動的 ACML(Korel と Laski) でのスカラー変数のマークアップ

```
int a;

<variable>
   <name>a</name>
   <address>0x22fd44</address>
   <size>4</size>
   <value>1</value>
</variable>
```

図 4.13: 動的 ACML(Agrawal) でのスカラー変数のマークアップ

```
int a[10];

<variable>
   <name>a</name>
   <offset>1</offset>
   <address>0x22fd44</address>
   <size>4</size>
   <value>1</value>
</variable>
```

図 4.14: 動的 ACML(Agrawal) での配列のマークアップ

```
struct test{
   int x;
   int y;
};

struct test a;

<variable>
   <name>a</name>
   <offset>x</offset>
   <address>0x22fd44</address>
   <size>4</size>
   <value>1</value>
</variable>
```

図 4.15: 動的 ACML(Agrawal) での構造体のマークアップ

```
10 : while(i < 1){
11 :     ...;
12 :     ...;
      }
```

図 4.16: 制御命令 `while` の例

```
<loop statement_name='while'>
   <ref>
      <variable>
         <type>int</type>
         <name>x</name>
         <value>0</value>
      </variable>
   </ref>
   <truth>true</truth>
   <block>
      <line id='a8' number='11'>...</line>
      <line id='a9' number='12'>...</line>
   </block>
</loop>
```

図 4.17: `while` 文の例 (Korel と Laski)

```
<iteration description='while' boolean='true'>
   <conditional>
      <variable access='reference'>
         <name>x</name>
         <address>0x22fd44</address>
         <size>4</size>
         <value>0</value>
      </variable>
   </conditional>
   <line id='a8' number='11' depend='a7'>...</line>
   <line id='a9' number='12' depend='a7'>...</line>
</loop>
```

図 4.18: `while` 文の例 (Agrawal)

4.3.5 動的 ACML の特徴

我々の実現した動的 ACML の特徴は以下の通りである。

- 情報取得の粒度
  動的スライシングは、ステートメントや変数など細粒度の情報を必要とする技術である。しかし、実行系列は、先でも述べたとおり、サイズが大きくなりやすい。よって、必要となる情報だけ細粒度で取得し、必要の無いものは粗粒度で、もしくは取得しないことが必要となる。我々の定義した動的 ACML では、変数に関する情報 (型情報、シンボル名、アドレス値、変数値、サイズなど) を細粒度で取得し、構文情報など、静的に取得できる情報は、粗粒度とした。粗粒度な例としては、関数情報を取得しない部分や、制御文以外のステートメントは、同一視するなどがある。

- コンパクトさ
  実行系列は前述の通り、情報のサイズが大きくなりやすい。また、XML 文書もタグ付けを行う関係上データサイズが増える。また、動的な情報はそれぞれの情報に寿命があり、全ての情報を取得することは困難であるし、無駄なことである。さらに、データスキーマが巨大であると、そのものの認識に時間がかかり、開発効率に影響する。よって、我々は、動的 ACML をコンパクトに実現するように設計した。基本的には、変数の動的スライシングに必要な情報を、差分を取ることを考える。また、変数は実行時点で必ず使われるものではないため、利用 (使用、定義) した変数のみの情報を取ることとした。実際に定義した動的 ACML による XML 文書は、ループ命令の回数に比例するため一概には言えないが、ソースコードの約 20 倍となっている。これは XML を利用していることを考えると許容範囲であると考える。

- 認識のしやすさ
  動的 ACML を、実行系列抽出部分開発者、依存解析ツール開発者がデータスキーマとして利用することを前提としている。動的 ACML が複雑な構造を持っていて、かつ巨大であり、わかりにくいものであった場合、両開発者共にデータスキーマの理解にかなりの時間を必要としてしまう。そうなると開発コストの増加にもなりかねないため、できる限り認識のしやすさが必要となる。そこで、動的 ACML は、基本として実行時点をインデックスとした表構造を持っている。実行時点ごとに、使用、定義した変数についての情報を持っていて、変数の情報は、木構造である。また、制御命令は動的スライシングで特別な意味を持つため、個別のタグ付けをし、認識の容易性を提供している。

- アクセスのしやすさ
  動的 ACML では、必要な情報へのアクセスのしやすさも提供しなければならない。

なぜならば、サイズがコンパクトで認識しやすい構造を持つデータスキーマであっても、アプリケーション側から煩雑な手順を経なければ情報にアクセスできない場合は、やはり開発コストが増加してしまう為である。例えば、制御命令では、静的な構文情報でわかる制御範囲という情報が必要である。これは動的な情報からも解析可能であるが、情報抽出時点で明らかになっている情報のため、動的 ACML の構造に埋め込んだ。実際の方法としては、条件命令のタグに、子要素として実行時点をネストさせている。これにより、動的 ACML 文書を目視する場合でも、制御範囲がわかりやすくなる。さらに Agrawal 用の動的 ACML では、動的スライシングは、「制御命令から制御依存関係を求める」のではなくて、「ある実行時点からその制御関係を持った制御命令を求める」と言うことを考慮し、ネストしている子要素から制御命令に向けてのリンクを持たせてある。これにより、動的 ACML 文書の時点で、制御依存はグラフ構造で保持されていることになる。

第5章　XMLを利用した動的スライサの実装

5.1　概要

我々は実現した動的スライサ用のデータスキーマである動的 ACML を用い、二種類の動的スライサの実装を行った。実装では ANSI C のサブセットを設定した。二種類の動的スライサの開発目的は以下の通りである。

- XML を用いた動的スライサの有用性
  XML を用い、動的スライサの開発コスト削減につながるかどうか、最小限度の ANSI C のサブセットを設定し、有用性の検証を行った。
- より実用的な動的スライサ
  動的スライサは有用な技術である [8]。しかし、先に示した最小限度の ANSI C のサブセットでは、一般的なプログラムにはほとんど対応できず、利便性が極めて悪い。よって、我々の考えるゴールである、フルセットの ANSI C への対応の第一段階としてポインタの導入を考える。ポインタを選択した理由は 5.2.1 節で詳細を示す。

5.2　ANSI Cのサブセット

我々は、フルセットの ANSI C に対応した動的スライサをゴールと考えている。しかし、ANSI C の情報は多種多様であり、全てを考慮するのは困難である。そこで、我々は研究の第一歩として ANSI C のサブセットを対象とすることとした。以下に、サブセットの詳細を示す。

5.2.1　サブセットの要素

- スカラー変数
  動的スライシングは、変数間の依存関係を解析する技術である。動的スライサの実装で、実装の困難な部分とアルゴリズム自体の困難な部分を場合分けし、動的スラ

イサの本質的な有効性と難しさを残したまま簡略化するため、我々は、スカラー変数の int 型のみを用いることとした。

- 制御文
  制御文は、制御依存関係に関連する重要な項目である。動的スライサの本質的な有効性を確認するためには必須の項目であると考える。また、制御文の条件部分において、簡単のために比較演算子のみの使用とする。

- ライブラリ関数
  ライブラリ関数は、プログラムに必要であるが、動的スライシングでは、変数の使用と定義がわかって、ライブラリ関数内でバグがないという仮定をすれば、'+' や'=' などの演算子と同じ扱いができるものと考える。よって、我々は、サンプルプログラムを作成する上で必要最小限 ("printf"や"fgetc"など) であるものを特別視し、他のライブラリ関数を利用しないこととした。

- システムコール
  一般的に、システムコールは環境に依存していることが多くセマンティクスが言語外にある。また、システムコール自体のソースコードも入手できないことが多い。よって、今回の実験には、一切のシステムコールを排除した。

- 関数呼び出し
  ANSI C では、ある規模のプログラムではユーザ定義関数を一般的であると考える。また、節 4.3.1 にあるように、ANSI C プログラムで、関数呼び出しはバグの要因になりやすい点である。これは、動的スライシングにとって重要な要素であると考える。しかし、関数呼び出しでのバグには、ポインタ演算を伴っていることが多く、ポインタ演算を含まずに関数呼び出しに対応するメリットが少ないと考える。よって、我々は、関数呼び出しを今回の実験仕様からはずすこととした。

- ポインタ演算
  ポインタ演算は、C 言語でプログラムを作るにあたってとても有用な方法だが、それとともに、バグを引き起こす大きな要因でもある。ポインタの計算ミスから起こるバグは、時にセグメンテーションフォールトなどの重大なエラーを引き起こすのに、発見はかなり困難を要する。それは、ポインタ値が実行しないと定まらない点が大きく寄与していると考える。これは、動的解析の代表的なテストケースとして考えられる。動的スライシングによってポインタの依存関係を解析することは工学的にも有用であると考える。しかし、一般的に C 言語での正確なポインタ解析は静的では困難とされている [9]。これは、C 言語が他の言語に比べて自由なポインタ操作を認めていることと、ハードウェアや処理系に依存したコードを書きやすいことがある。例えば、図 5.1 は初期化していないポインタへの利用で、図 5.2 は NULL ポインタへの代入の例である。

```
int *p

*p = 3;
```

図 5.1: 初期化されていないポインタの使用

```
int a, *p;

p = NULL;
*p = 12;
```

図 5.2: NULL ポインタへの代入

- 構造体と配列
  構造体と配列は、それぞれ特徴を持つ変数である。構造体は各々にメンバを持ち、それぞれのメンバが別の変数の特徴を持つことが多い。また、構造体のメンバに構造体への参照ポインタを持たせることによって、リスト構造として利用する方法は、一般的によく使われる方法である。配列は、インデックスによる、部分アクセスが可能である。このインデックスに変数を用いると、静的解析では正確な解析ができない。例えば、配列 a、整数型i、j があるとき、a[i] とa[j] が同一であるかどうかは、静的には解析できない。これらは動的スライシングの有用性を検証するためには重要な要因であると考える。我々は、この要素を Agrawal のアルゴリズムを使う依存解析ツールに含める。

5.2.2 定義した ANSI C のサブセット

これらを踏まえ、以下に実際に使用した ANSI C のサブセットを示す。

- Korel と Laski のアルゴリズム
  - 変数は整数型の変数のみを使用する。配列、構造体、その他の変数は使用しない。ただし、ファイル識別子を使用する。
  - 制御文は while 文と if-else 文のみで、条件命令には、比較演算子のみを使用する。
  - 関数呼び出し、ポインタ演算は含まない。
  - ライブラリ関数、システムコールは基本的に使用しない。
    ただし、”printf”、”fgetc”、”fopen”、”fclose”は、特別視し使用する。
- Agrawal のアルゴリズム
  - 変数は、整数型、配列、構造体とポインタとする。
  - 制御文は while 文と if-else 文のみで、条件命令には、比較演算子のみを使用する。
  - 関数呼び出しは行わない。
  - ライブラリ関数、システムコールは使用しない。
    ただし、scanf、printf は、特別視し、使用する。

また、両方のアルゴリズムで、簡単のため、一つの行に、式文は一つとした。

5.3 依存解析ツールの実現

我々は、実装実験として、二種類のアルゴリズムをそれぞれ用いた、依存解析ツールの作成を行った。用いたアルゴリズムは以下の二種類である。

- Korel と Laski のアルゴリズム [4]
- Agrawal のアルゴリズム [7]

5.3.1 動作状況

各アルゴリズムについての動作状況は以下の通りである。

- Korel と Laski のアルゴリズム
  スライス基準 (行番号, 変数名) を基に、動的 ACML 文書に対し、依存関係のある実効時点を抽出する。以下に、動作結果を例示する。図 5.3 に対して、入力値”hoge2”を与えて抽出した実行系列に、スライス基準 (31, sum) で実行した場合を図 5.6 で、それを基にしたスライスは図 5.4 で示す。また、同一の実行系列に、スライス基準 (30, min) で実行した場合の結果を図 5.7、それを基にしたスライスを図 5.5 で示す。
  さらに、図 5.9 のプログラムに対し、入力値”$n = 3, x = [-4, 3, -2]$”を与えて実行した実行系列に対し、スライス基準 (19, z) で解析した結果を、図 5.8 で、スライス[1]を図 5.10 で示す。

- Agrawal のアルゴリズム
  Agrawal のアルゴリズムに基づき、動的依存グラフを抽出し、スライス基準に従って動的スライサを解析する。現在の実装状況は、75%程度で、動的依存グラフの抽出まで完了している。ただし、アルゴリズム簡略化のため、Cell の交差のチェックは行っておらず、「先頭アドレスが同じであれば、必ず完全一致し、先頭アドレスが異なれば交差しない。」としている。

5.3.2 開発コストに関する客観的データ

この節では、実装実験を通して得た客観的データを基に、開発コスト削減や、その他の要因にどのような影響をもたらしているか検証を行う。

[1]現在は、実装上の小さなバグのため、本来入っているはずの 10 行目と 11 行目が抜けている

```
 1 : #include<stdio.h>
 2 :
 3 : int main(int argc, char *argv[]){
 4 :   FILE *fp;
 5 :   int max, min, sum, c;
 6 :
 7 :   fp = fopen(argv[1], "r");
 8 :    if (fp == NULL) {
 9 :     printf("%s can't be opened\n", argv[1]);
10 :     exit(1);
11 :    }
12 :
13 :    c = fgetc(fp);
14 :    max = c;
15 :    min = c;
16 :    sum = c;
17 :    c = fgetc(fp);
18 :    while(c != EOF){
19 :       if (c > max)
20 :          max = c;
21 :       if (c < min)
22 :          min = c;
23 :       sum = sum + c;
24 :       c = fgetc(fp);
25 :    }
26 :    fclose(fp);
27 :
28 :
29 :    printf("max:%c\n", max);
30 :    printf("min:%c\n", min);
31 :    printf("sum:%d", sum);
32 :
33 :    return 0;
34 : }
```

図 5.3: テストケース 1 のソースプログラム

```
 1 : #include<stdio.h>
 2 :
 3 : int main(int argc, char *argv[]){
 4 :   FILE *fp;
 5 :   int max, min, sum, c;
 6 :
 7 :   fp = fopen(argv[1], "r");
 8 :
 9 :
10 :
11 :
12 :
13 :    c = fgetc(fp);
14 :
15 :
16 :    sum = c;
17 :    c = fgetc(fp);
18 :    while(c != EOF){
19 :
20 :
21 :
22 :
23 :       sum = sum + c;
24 :       c = fgetc(fp);
25 :    }
26 :
27 :
28 :
29 :
30 :
31 :    printf("sum:%d", sum);
32 :
33 :
34 : }
```

図 5.4: テストケース 1 のスライス基準 (31, sum) についてのスライス

```
 1 : #include<stdio.h>
 2 :
 3 : int main(int argc, char *argv[]){
 4 :   FILE *fp;
 5 :   int max, min, sum, c;
 6 :
 7 :   fp = fopen(argv[1], "r");
 8 :
 9 :
10 :
11 :
12 :
13 :    c = fgetc(fp);
14 :
15 :    min = c;
16 :
17 :
18 :    while(c != EOF){
19 :
20 :
21 :       if (c < min)
22 :          min = c;
23 :
24 :
25 :    }
26 :
27 :
28 :
29 :
30 :    printf("min:%c\n", min);
31 :
32 :
33 :
34 : }
```

図 5.5: テストケース 1 のスライス基準 (30, min) についてのスライス

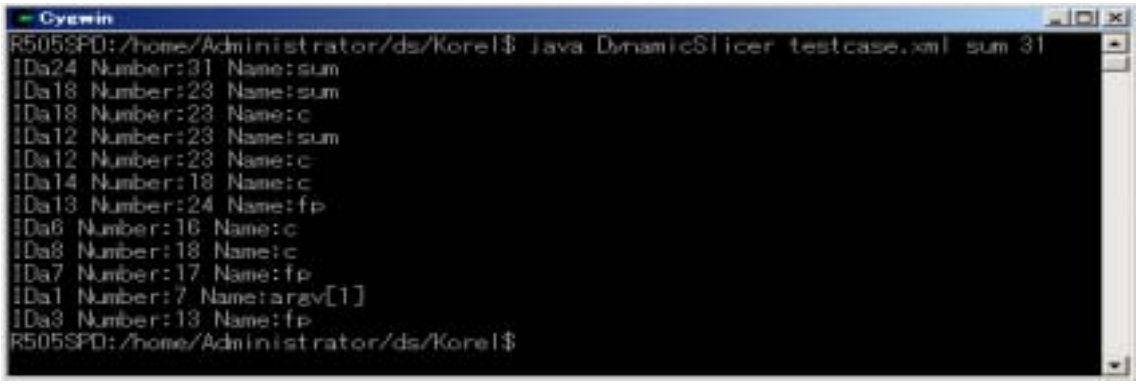

図 5.6: スライス基準 (31, sum) でテストケース 1 に対する解析結果

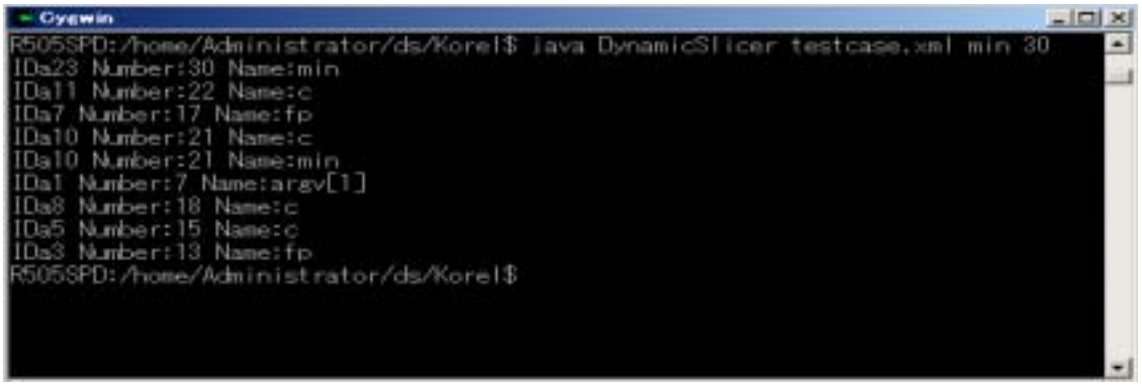

図 5.7: スライス基準 (30, min) でテストケース 1 に対する解析結果

図 5.8: スライス基準 (19, z) でテストケース 2 に対する解析結果

```
 1 : #include<stdio.h>
 2 :
 3 : int fun1(int a);
 4 : int fun2(int a);
 5 : int fun3(int a);
 6 :
 7 : int main(void){
 8 :    int n, i, x, y, z;
 9 :
10 :    scanf("%d", &n);
11 :    i = 1;
12 :    while(i <= n){
13 :       scanf("%d", &x);
14 :       if (x < 0)
15 :          y = fun1(x);
16 :       else
17 :          y = fun2(x);
18 :       z = fun3(y);
19 :       printf("%d", z);
20 :       i++;
21 :    }
22 :    exit(0);
23 : }
24 :
25 : int fun1(int a){
26 :    return a;
27 : }
28 :
29 : int fun2(int a){
30 :    return a;
31 : }
32 :
33 : int fun3(int a){
34 :    return a;
35 : }
```

図 5.9: テストケース 2 のソースプログラム

```
 1 : #include<stdio.h>
 2 :
 3 : int fun1(int a);
 4 : int fun2(int a);
 5 : int fun3(int a);
 6 :
 7 : int main(void){
 8 :    int n, i, x, y, z;
 9 :
10 :
11 :
12 :    while(i <= n){
13 :
14 :       if (x < 0)
15 :          y = fun1(x);
16 :
17 :
18 :       z = fun3(y);
19 :       printf("%d", z);
20 :       i++;
21 :    }
22 :
23 : }
24 :
25 : int fun1(int a){
26 :    return a;
27 : }
28 :
29 : int fun2(int a){
30 :    return a;
31 : }
32 :
33 : int fun3(int a){
34 :    return a;
35 : }
```

図 5.10: テストケース 2 のスライス基準 (19, z) についてのスライス

サンプル 1 スライス基準 (31, sum)	305.94
サンプル 1 スライス基準 (30, min)	288.78
サンプル 2 スライス基準 (19, z)	297.22
サンプル 3	354.11

表 5.1: 実行時間

	コード数	完成度
Korel と Laski のアルゴリズム	約 300 行	99%
Agrawal のアルゴリズム	約 250 行	75%

表 5.2: 依存解析ツールの開発コスト

実装時間と実行時間

- 実装時間
  実装時間は、各アルゴリズム、各々一人の開発者で約 2 週間であった。これは、動的スライサの実装としてはきわめて短いものだと考える。JAIST の川島、権藤 [31] によると、静的な構文情報をタグ付けして出力するトランスレータの実装に 2ヶ月、静的スライサ、クロスリファレンサなどの静的な CASE ツールの実装に各々約 2 週間である。これらと比較すると、小さな実装実験に限り開発コストの削減を確認した。

- 実行時間
  今回の実装実験では、Sun Microsystems 社の J2SDK1.4[29] で Java の実行環境を用いて実装を行った。動的スライサの実行時間を表 5.1 に示す[2]。

コード量

実際のコード量を表 5.2 に示す。
動的スライサの比較対照が無いので、一概には言えないが、動的依存解析ツールとして比較的コンパクトに自走できたものと考える。これは、データスキーマが理解しやすく、動的スライサのアルゴリズムにあった形で設計されていることで、情報を取り出すための操作が比較的少なくて済んだためだと考える。Agrawal の実装の方がコード量が少ないが、これは実装の完成度によるものである。

[2]単位はミリ秒で、20 回計測し、上下 1 回を省いた 18 回の平均値とする

	ソースコード	CSV 形式	動的 ACML
サンプル 1	505	1271	8198
サンプル 2	404	917	7706
サンプル 3	456	1449	7997

表 5.3: 依存解析ツールの開発コスト

XML 文書のサイズ

テストケースとしたサンプルプログラムの実行系列のサイズを考える。単位は Byte とする。実行系列はループ命令の回数に比例して大きくなる可能性があるため一概には言えないが、ソースコードに対し、XML 文書化することで約 20 倍になっていることがわかる。また、CSV 形式のファイルと比較すると 5.5 倍から 8.5 倍程度に収まっていることがわかる。これは XML 文書かしたことを考えると、許容範囲のサイズ増加だと考える。

第6章　結果

6.1　動的スライサの開発コスト

我々は、動的スライサの開発コスト削減に、XML が利用できるか実装実験を行った。実装実験は、研究の第一歩として、ANSI C のサブセットに対するもので、最も基本的なアプローチである Korel と Laski のアルゴリズムを利用した動的スライサと、ポインタを考慮した Agrawal のアルゴリズムを利用した動的スライサの二点で行った。

6.1.1　実装のコスト

実際に実現した動的スライサの、依存解析ツールは、開発者一人がそれぞれ約二週間で行った。これは非常に短期間ですんだと考える。

6.1.2　コスト削減の要因

第 5 章の実装実験から、開発コスト削減の要因となったと考えられる項目を以下に示す。

- 中間データをテキストファイルとして導入したことによる要因
  - 実行系列の抽出と解析の分離
    我々は、実行系列を中間データとして、テキストファイルである XML 文書に書き出した。これにより、我々の小規模な実験では、実行系列の抽出をする C インタプリタと、実行系列を解析し、動的スライスを出力する依存解析ツールの実装を分離することができた。よって、依存解析ツールの開発では動的スライシングの本質的な部分の開発に専念でき、開発コストの削減につながったと

	作成期間	使用技術	コード数
Korel と Laski のアルゴリズム	一人で約二週間	DOM	約 300 行
Agrawal のアルゴリズム	一人で約二週間	DOM	約 250 行

表 6.1: 依存解析ツールの開発コスト

考える。

- テストケースの手作業での作成
  中間データをテキストファイルとすることで、人間に認識しやすい形となった、これにより、実行系列のテストケースを手作業で作り出すことができ、実行環境が存在しない状態でも、依存解析ツールの動作テストが可能となった。

- 実行系列の目での確認
  実行系列がテキストファイルであるため、依存開発ツールの作成時に、直接 XML 文書ファイルを目で確認しながらの作業が可能であった。これはメモリ上に展開されたデータや、バイナリ形式で書き出されたデータファイルでは容易にはできない。なぜなら、メモリ上のデータは、開発者が頭の中で展開したデータが、実際のデータと一致してるという保証が無く、これが一致していないとき、新たなバグの要因になりかねない為である。また、バイナリファイルでは、一般的にそのバイナリファイルの開発者以外には理解が難しく、他の開発者がそのデータの意味を理解するのに非常にコストがかかるため総合的に見て、開発コストが増加することが考えられるからである。

- 実行系列を XML 文書で表現したことによる要因

  - 実行系列の構造に意味を付加
    XML 文書は構造化文書である。全ての要素にタグによって情報の付加がされている。例えば、”0x22fd44”と言うテキストがアドレスを表すとき、CSV 形式だと、開発者はアドレス値だということはわかるが、これがポインタ変数のアドレス値 (つまり `&p`) なのか、ポインタ変数の持つ値 (`p`) なのか、判断する情報がない。しかし、XML 文書であればそのテキストの前に <> で囲まれたタグが存在する。これによって、新たな情報が得られ、それにより、情報の意図を推測することが可能である。

  - 字句解析、構文解析を XML 技術で補完
    XML 文書は構造化文書であるが、テキストファイルに一度、保存するので、字句解析、構文解析が必要である。しかし、XML には、XML パーサが存在する。XML パーサは Java や C などの独自プログラムに組み込むことが可能である。一般的な CSV 形式などではツール開発者が情報取得のための字句解析、構文解析器を作成しなければならず、これはツールの本質でないにもかかわらず大きな負担であった。しかし、XML では、この機能が有志により作られており、多くのものはフリーで手に入るため、この部分のコストの削減が可能である。

- DOM の操作による文書操作コストの削減
  DOM は、XML 文書の操作に使われる汎用的な技術の一つである。DOM は XML 文書の木構造を柔軟に探索可能である。この木構造の操作は直感的にわかりやすく、DOM 自体も汎用的な技術であるため、操作習得は容易である。

- 適切なデータスキーマの設計
  - 実行系列をコンパクトに提供
    プログラムの実行時情報には、構文情報や、変数の値、シンボル名ループの回数、アドレス値、型情報など多種多様な情報が存在する。しかし、これら全てを実行系列として保持すると、明らかにサイズの増加が起きるためデータの取捨択一は必要であった。例えば、構文情報を取得すると、四則演算の加算と減算だけでも別の式として扱わなければならない。しかし、これは動的スライシングでは必要としない情報である。こういった項目を考慮しながら必要な情報を残して、かつ、不必要な情報を削除したデータスキーマの設計を行った。
  - 抽出側、解析側、両者に処理しやすい形
    本研究の第一段階として定義した動的 ACML は、動的スライサの開発コストを削減する観点から、依存解析ツール側の使いやすいものを第一に考えて定義した。しかし、それにより、データ抽出側の開発効率や、実行効率が著しく悪くなっては意味がない。そこで、両者から扱いやすい形を持たせる努力をした。例えば、付録 A の動的 ACML では定義、使用で、変数をまとめている。しかし、一般的に実行環境は抽象構文木の木構造にしたがって式を評価するため、必ずしも定義、使用がまとまっているとはいえない。その場合、動的 ACML にあわせた XML 文書を抽出するために実行環境内で情報の保持が必要になる。しかし、付録 B で定義、使用を変数の属性としたところ、解析側でよく t 買う情報であるため属性を取得するという作業が一つ増えることとなった。このように、データ抽出側と、解析側の両方に使いやすい形でのデータスキーマ定義を行った。
  - 単純な構造による理解しやすい形
    データスキーマを単純な構造で理解のしやすい形で提供した。データスキーマが単純な構造であると、その理解が容易であり、開発の時間が削減される。データスキーマが理解できなければ、開発はできないであろう。しかい、構造が単純すぎても、情報が欠落するか、または、解析に時間がかかってしまう。例えば、図 6.1 の様なデータ構造の場合、図 6.2 の様に、必要な情報が欠落してしまう。また、図 6.2 の様に、無理やり情報を詰め込むと、解析に時間を要してしまう。よって、適切な情報を取得しつつ、できるだけ単純なデータ構造を実現した。これにより、データスキーマが理解しやすくなり、開発コスト削減につながったと考える。

```
<!ELEMENT root (line)+>
<!ELEMENT (#PCDATA)>
```

図 6.1: 単純なデータ構造の DTD

```
<root>
   <line>1</line>
   <line>2</line>
   .
   .
   .
</root>
```

図 6.2: 情報が不足している XML 文書

```
<root>
   <line>1,10,x,def,int,10</line>
   <line>2,11,x,ref,int,10,y,def,int,10</line>
   .
   .
   .
</root>
```

図 6.3: 解析に時間のかかる XML 文書

6.2 検討すべき点

実装実験より、以下の検討すべき点を見つけた。

6.2.1 ANSI C サブセットに対しての制限の緩和

我々の目指すゴールは、ANSI C のフルセットに対応した動的スライサである。よって、制限の緩和は必要な項目である。我々は以下の項目の緩和の優先度が高いと考えている。

全てのポインタ演算への対応

ANSI C は、自由なポインタ操作を記述できる言語である。このため全てのポインタへの対応は大変困難である。しかし、一般的に記述される、C 言語のプログラムは、ポインタ演算の間違いなどからバグを引き起こすことは少なくない。よって、ポインタ演算は重要な項目と考える。

関数呼び出しへの対応

C プログラムにおいて、ユーザ定義関数を使わないプログラムは少ない。関数呼び出しは一般的に使われる手法である。また、4.3.1 節にあるようにバグの要因となることが多い。また、動的スライシングでポインタ解析と並び、よく研究されている項目である。これより、重要度は高いと考える。

変数型

現在は、整数型、構造体、配列、ポインタ型のみとしている。一般的なプログラムを扱うためには、文字型や文字型の配列である文字列、倍精度整数型、単精度実数型など多くの型を扱うべきであると考える。

6.2.2 XML 文書のサイズ

実行系列を XML 文書にするとサイズは増える。現在は、ハードウェアの進歩により無視できる範囲のサイズ増加であると考えるが、制限の緩和に伴い、取得する情報が増えることにより、無視できないサイズになる可能性がある。このような問題に対処するアプローチ [34] も研究されている。このアプローチは、節点集約と言う方法を提案し、情報の共有による情報量の削減や、節点の減少による計算量の削減が期待されている。それらを適用することも考えられる。

6.2.3 DTD の利便性

動的 ACML は、XML 文書のデータスキーマとして働く。我々の研究で動的 ACML は DTD で定義した。DTD のよい点として、拡張 BNF で記述できるため、プログラムの情報を自然な形で表現することが可能である。また、SGML で使われていた経緯から成熟した技術であるので、DTD を用いての検証を行える XML パーサが多く存在していることがあげられる。しかし、この古くからあるという点は欠点でもある。その一つはデータ型の問題である。DTD では一部を除いてデータ型は全てテキストである。例えば動的スライサで、行番号を保持するところは整数型、ポインタアドレスを保持するところは 16 進数型など、データ型が多く存在すると、アプリケーション側での不正なデータのチェックにかかるコストが削減できる。

6.2.4 DOM の利便性

DOM は、XML 関連技術の中でも広く認知されており使いやすい技術である。操作も簡単であり習得にも時間がかからない。また、XML 文書を新規に作り出すことや、要素、属性を書き換えたりなども行うことができる。しかし、DOM は、XML 文書を、DOM ツリーに展開する。これは、メモリ空間に保持されるため、大きなメモリ空間を必要とする。実行系列が大きくなった場合、DOM では扱えない可能性もある。さらに、DOM は基本的な操作しかもっていないので、同じ操作を反復で行うことが多々存在する。これは、後述するライブラリの必要性と他の XML 関連技術の利用につながる。

6.2.5 ライブラリの必要性

我々は実装実験において DOM を利用した。DOM は、操作が簡単で習得に時間がかからず、XML 文書の木構造をなぞる操作性は直感的にわかりやすい。しかし、複雑な、もしくは大きな XML 文書を扱う場合、同じような作業が何度も必要となることがある。例えば、図 6.4 で、現在のカレントが 1 のとき、4 にアクセスする方法は、1 の最初の子である 2 にアクセスし、2 の次の子である 3、3 の次の子である 4 と、アクセスすることになる。これらは非常に煩雑であるが、よく起こる操作でもある。これらをの定型的な操作をライブラリとして提供することによって、さらに開発コストが削減できると考える。

6.2.6 他の関連技術の利用の可能性

我々は今回の実装実験に、XML パーサと DOM という XML 関連技術を用いた。DOM は、複雑な XML 文書の検索は得意としない。例えば、図 6.5 で、「*variable* を要素として二個持つ要素を探す」といった操作を DOM でする場合、アプリケーションで処理する部分が大きい。こういった場合、XSLT や XPaht を使えば、はるかに容易に検索することも

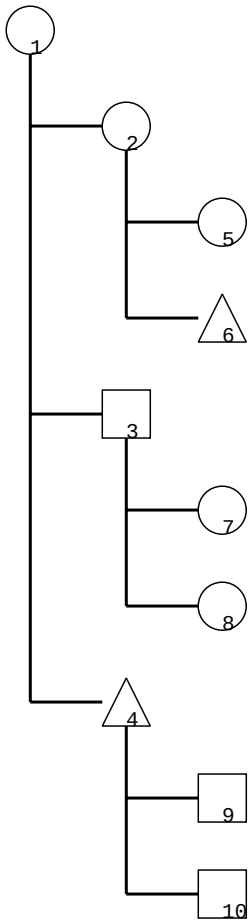

図 6.4: 木構造の例

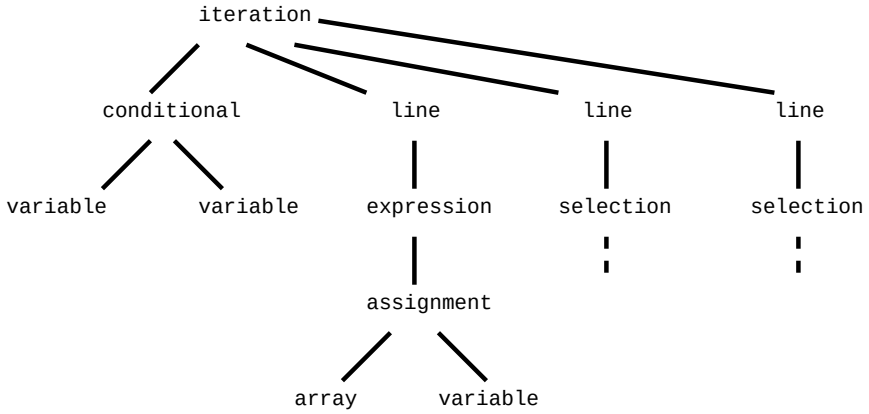

図 6.5: 動的 ACML 構造の一部

可能である。例えばこの時、XPath だと "$//*[\texttt{count}(\texttt{variable}) = 2]$" という式で検索ができる。

第7章　おわりに

7.1　まとめ

我々は、動的スライシング技術は工学的に有用であるのに、動的スライサが普及しない要因として、開発コストの高さに注目した。動的スライサの開発コストは、実行環境が必要なことや、実行系列の情報取得の難しさ、実行系列を解析するときの字句解析や構文解析などの動的スライシング技術とは本質的に関係のない部分のコストの高さなどが影響していると考えた。これらの要因に対して、中間データとして実行系列を XML でマークアップすることにより、コスト削減を目論み、動的 ACML を提案した。本研究では、自由なメモリアクセス、副作用による問題、エイリアスによる問題などによりバグとなる要因が多く、保守コストが高いため、動的スライサの利用が望める ANSI C をターゲット言語とした、ANSI C に対する動的スライサを提案した。この動的スライサの構成要素は以下の 3 つである。

1. XCI
   ANSI C を実行し、動的 ACML に基づいた XML 文書を中間データとして抽出する C インタプリタ
2. 動的 ACML
   実行系列を XML で表現するためのデータスキーマ
3. 依存解析ツール
   動的スライシングのアルゴリズムを適用し、動的 ACML に基づいた XML 文書からスライスを抽出する依存解析ツール

我々は、実装実験として、ANSIC のサブセットを定め、二種類の依存解析ツールの実装を行った。この二種類の実装は以下の理由に基づいて作成した。

- 動的 ACML を用いた動的スライサの開発コストに対する有用性を検証する。
- より一般的なソースプログラムに対して動的スライサが適用可能であるかどうか検証する。

これらの実験において、ツール開発者は、一人でそれぞれ約二週間ですんだ。これは動的 ACML を用いることで、本来同時に開発しなければならない C インタプリタの開発コストを分離できたことを意味する。これらの実装実験では、小さな ANSI C のサブセットに対し、XML を用いることによって動的スライサの開発コストの削減を確認できた。ま

た、これら実装実験で得られた経験から、動的 ACML の改良点や、今回使用した XML 関連技術の不満な点、他の XML 関連技術の適用の可能性、定型作業のライブラリ化など検討すべき項目を発見することができた。

7.2 結論

我々の行った実装実験で、動的スライサの開発効率を上げることの要因となったものは以下のとおりと考える。

- 中間データをファイルとして導入
  - 実行系列の抽出と解析の分離
  - テストケースの手作業での作成
  - 実行系列の目での確認
- 実行系列を XML で表現
  - 実行系列の構造に意味を付加
  - 字句解析、構文解析を XML 関連技術で補完
  - DOM の使用による文書操作のコスト削減
- 適切なデータスキーマの設計
  - コンパクトに実行系列を提供
  - 抽出側、解析側、両者に処理しやすい形
  - 単純な構造による理解しやすい形

また、実装実験から、検討するべき点は以下の通りである。

- XML 文書のサイズ
- 動的 ACML の汎用性
- ライブラリの必要性

7.3 今後の課題

我々は、研究の第一段階として ANSI C のサブセットに対して、動的スライサの実装実験を行った。この研究の今後の課題を以下に示す。

- 制限の少ない動的スライサ
  現在の解析ツールは数々の制限を付けている。実用的な動的スライサを目指すには制限の解除は不可欠である。

- 関数呼び出しの対応
  ANSI C プログラムは、一般的に関数呼び出しを行う。よって、これに対応することは上記項目と同じように、実用的な動的スライサに必要と考える。

- ライブラリの作成
  DOM での XML 文書操作は、同じような作業を何度もしなくてはいけないことが多い。定型的に扱う操作をライブラリとして提供することによって、更なる開発コストの削減を望めると考えた。

- Java や SML など、他言語への対応
  どんなプログラムでも、バグは起きる。いろいろな言語に対応することによってツールの研究も盛んになると考える。

- ACML との連携
  現在は、動的情報に絞って情報を取っている。そこで、静的な CASE ツールプラットフォームである ACML と連携すれば、静的な情報も使う動的ツールにも対応が可能である。

- GUI[1] ツールの導入
  動的スライサを手軽に利用する場合は、GUI も必要であると考える。

- 他の動的 CASE ツールへの利用
  動的 ACML を利用することによって、プロファイラーやプログラムシュミレータなどの他の動的 CASE ツールにも利用できれば、動的な CASE ツールプラットフォームができると考える。

[1] Graphical User Interface

謝辞

本研究を進めるにあたり、最後まで熱心にご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、修士論文をともに戦い、励ましあった仲間たちに感謝します。

付録 A

```
<!ELEMENT hist_root (line)+>
<!ELEMENT line (loop | call | branch | assign)>
<!ELEMENT loop (ref?, truth, block?)>
<!ELEMENT branch (ref?, truth, block?)>
<!ELEMENT call (def?, ref?)>
<!ELEMENT assign (def?, ref?)>
<!ELEMENT block (line)+>
<!ELEMENT def (variable)*>
<!ELEMENT ref (variable)*>
<!ELEMENT variable (type, name, value)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT truth (#PCDATA)>
<!ATTLIST line
    id ID #REQUIRED
    number CDATA #REQUIRED
>
<!ATTLIST branch
    statement_name (if | switch) #REQUIRED>
<!ATTLIST loop
    stetment_name (while | for | until) #REQUIRED>
<!ATTLIST call
    function_name (fgetc | fopen | fclose | printf) #REQUIRED>
```

付録 B

```
<!ELEMENT root (line)+>
<!ELEMENT line (selection | iteration | expression)>
<!ELEMENT iteration (conditional, line*)>
<!ELEMENT selection (conditional, line*)>
<!ELEMENT expression (condistional | assignment)>
<!ELEMENT conditional (variable | struct | array)*>
<!ELEMENT assignment (variable | struct | array)*>
<!ELEMENT variable (name, address, size, value)>
<!ELEMENT struct (name, offset, address, size, value)>
<!ELEMENT array (name, offset, address, size, value)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT offset (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT size (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ATTLIST line
    id ID #REQUIRED
    number NMTOKEN #REQUIRED
    depend IDREF #IMPLIED>
<!ATTLIST variable
    access (definition | reference) #REQUIRED>
<!ATTLIST array
    access (definition | reference) #REQUIRED>
<!ATTLIST struct
    access (definition | reference) #REQUIRED>
<!ATTLIST selection
    description (if | switch) #REQUIRED
    boolean (true | false) #REQUIRED>
<!ATTLIST iteration
    description (while | do_while | for) #REQUIRED
```

```
boolean (true | false) #REQUIRED>
```

参考文献

[1] Weiser, M., *"Programmers Use Slices When Debugging,"* CACM, Vol.25, No.7, July 1982, pp.446-452.

[2] Weiser, M., *"Program Slicing,"* IEEE Transaction on Software Engineering, Vol.SE-10, No4, July 1984, pp.352-357.

[3] Lyle, J. and Weiser, M. *"Automatioc bug location by program slicing."* In Proceedings of the Second International Conference on Computers and Applications, pp.877-883, 1987.

[4] Korel, B. and Laski, J., *"Dynamic Program Slicing,"* Infomation Processing Letters, Vol.29, No.10, Oct. 1988, pp.155-163.

[5] Korel, B. and Laski, J., *"Dynamic Slicing of Computer Programs,"* J. System Software, 13, 1990, pp.187-195.

[6] Agrawal, H. and Horgan, J. R., *"Dynamic program Slicing,"* ACM SIGPLAN Notices, Vol.25, No.6, June 1990, pp.246-256.

[7] Agrawal, H., DeMillo, R. and Spafford, E,. *"Dynamic Slicing in the presence of unconstrained pointers,"* Proc. of Symposium on Testing, Analysis and Verification, Oct. 1991, pp.60-73.

[8] F. Tip, " A Survey of Program Slicing Techniques, " TRCS-R9438, Univ. of Amsterdam, 1994.

[9] Marc, Shaprio. and Susan, Horwitz., *"The Effects of the Precision of Pointer Analysis,"* Static Analysis 4th International Symposium, SAS '97, Lecture Notes in Computer Science Vol 1302.

[10] G. A. Venkatesh, *"Experimental Results from Dynamic Slicing of C Programs,"* ACM Transactions on Programming Languages and Systems, Vol.17, No.2, 1995, pp.197-216.

[11] Steven P. Reiss and Manos Renieris *"Encoding Program Executions,"* International Conference on Software Engineering, May, 2001

[12] Anthony M. Sloane and Jason Holodsworth, *"Beyond Traditional Program Slicing,"* ACM SIGSOFT Software Engineering Notes , Proceedings of the 1996 international symposium on Software testing and analysis, May, 1996

[13] Hemant D. Pande, William Landi, *"Interprocedural Def-Use Associations in C Programs,"* Symposium on Testing, Analysis, and Verification 1991, pp139-153

[14] World Wide Web Consortium(W3C).
World Wide Web Consortium. http://www.w3c.org.

[15] World Wide Web Consortium(W3C).
Extensible Markup Language(XML)1.0(Second Edition).
http://www.w3.org/TR/REC-xml.

[16] World Wide Web Consortium(W3C).
HyperText Markup Language (HTML). http://www.w3.org/MarkUp/.

[17] World Wide Web Consortium(W3C).
Cascading Style Sheets. http://www.w3.org/Style/CSS/.

[18] World Wide Web Consortium(W3C).
Document Object Model(DOM). http://www.w3c.org/DOM/.

[19] World Wide Web Consortium(W3C).
The Extensible Stylesheet Language (XSL). http://www.w3.org/Style/XSL/

[20] World Wide Web Consortium(W3C).
XSL Transformations(XSLT) Version1.0. http://www.w3c.org/TR/xslt.

[21] World Wide Web Consortium(W3C).
XML Path Language(XPath) Version1.0. http://www.w3c.org/TR/xpath.

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

[23] Andreas Neumann, *The Functional XML Parser.*
http://www.informatik.uni-trier.de/ aberlea/Fxp/.

[24] Sun Microsystems, Inc. *JavaTM API for XML Processing (JAXP).*
http://java.sun.com/xml/downloads/jaxp.html.

[25] Microsoft Corporation. *Microsoft XML Core Services (MSXML).* http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000438.

[26] Apache XML Project *Xerces.* http://xml.apache.org/index.html.

[27] James Clark *an XML Parser in Java(XP).* http://www.jclark.com/xml/xp/index.html.

[28] David Megginson *Simple API for XML.* http://www.saxproject.org/.

[29] Sun Microsystems, Inc. *Java2 Platform, Standard Edition (J2SE).* http://java.sun.com/j2se/.

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

[31] 川島 勇人, 権藤 克彦, *XML* を用いた *ANSI C* のための *CASE* ツールプラットフォーム , 2002.

[32] 蔡 遠航, 権藤 克彦, 動的解析に適したインタプリタに関する研究, 電気関係学会北陸支部連合大会, 平成 12 年度.

[33] 福安 直樹, 山本 晋一郎, 阿草 清磁. 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

[34] 大畑 文明, 横森 励士, 西松 顯, 井上 克郎, スライス計算効率化のためのプログラム依存グラフの節点集約法, 電子情報通信学会論文誌 Vol.J84-DI No.7 July 2001

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

[36] David Hunter, Kurt Cagle, Dave Gibbons, Nikola Ozu, Jon Pinnock, Paul Spencer 著, 風工舎 訳, エキスパートから学ぶ *XML* 実践プログラミング, インプレス, 2001.

[37] PROJECT KySS/ビスケット株式会社 著, *XSLT* ⊕ *XPath* 実践マスター, ソフトバンク パブリッシング, 2002.

[38] インフォテリア株式会社 広瀬幸泰, *XML* の最新動向と *B2B* ソリューションの展望, 日本インターネット協会/JavaTM カンファレンス合同セミナー, 2000.

[39] 横井 与次郎, *DOM2* による *Java/XML* プログラミング, ソフトリサーチセンター, 2001.

[40] Mark Williams 社 編, 坂本 文/今泉 貴史 監訳, *C* 言語のための *ANSI C* 言語大辞典, パーソナルメディア, 1991.

[41] 中田 育男 著, コンパイラの構成と最適化, 朝倉書店, 1999.