修 士 論 文

Sapidを使った動的解析ツールの実装

指導教官　権藤克彦 助教授

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

篠井 隆典

2000年 2月 15日

要 旨

本稿では Sapid を用いた動的解析ツールの実装にについて述べる。Sapid を用いた C 言語のダイナミクスライサの実装をとおして、動的解析の問題点を探ると同時に Sapid の動的解析対する有用性について研究する。

目 次

図 目 次

表 目 次

第 1 章

はじめに

1.1 背景

ソフトウエア開発保守においてプログラムの実行状態に関する情報は重要である。例えば、実行時にはじめて明らかになるメモリリークなどの動的エラーのデバッグには、実行状態に関する情報は不可欠である。また、メトリックス、テストなどは問題は実行状態に関する情報を用いることにより、より正確な結果を得ることができる。特に、ポインタの追跡に関しては静的な解析では困難であるとの指摘が多い。したがって、動的なプログラム解析ツールは保守越すとの軽減に役に立つ。しかし、その実装はプログラムの規模が大きくなりやすく困難である。プログラムを実行するために、ソースコートの構文解析器、字句解析器をはじめとする様々な機能を実現する必要があるからである。しかも、これらの技術はすでに確立されており、その実装は本質的な問題ではないことが多い。それ故、これまでも様々な方法論が提案されているが、実装は多くない。実際に Purify[日本] などの有用なツールが存在する。しかし、これらのツールはメモリ管理に特化していたり、ツール間のデータの互換性がないなどの問題がある。

プログラム理解、デバッグ、テストなどのソフトウエア工学における問題に共通する解決法として、部分プログラムの抽出がある。プログラムからある特定の機能を実現している部分を抜き出すことにより、プログラムの構造を明確にしたり、実行パスを明らかにするのが狙いである。部分プログラムの抽出法の代表的なものにスライシング [Wei84] がある。スライシングはスタティックスライシングと、ダイナミックスライシング [HJ90] に分けることができる。このうち、ダイナミックスライシングは、入力に制限を設けることにより、比較的簡単なアルゴリズムにもかかわらず、正確にスライシングをすることが可能という特徴がある。また、スタティックスライシングでは困難との指摘があるポインタ追

跡においても、ダイナミックスライシングは有利である。実行パスが特定されるので、ポインタを追跡する範囲を著しく限定できるからである。しかし、動的な解析であるために実装が困難で、ある言語のサブセットや、特別に作られた簡易言語に対する実装が大部分である。ゆえに、実用的な言語にフルスペックで対応できるスライシングツールの実装が求められている。

細粒度リポジトリとしてSapid[山本95]が提案されている。これは、I-modelというモデルに基づいて作られたソフトウエアリポジトリである。I-modelに基づいたリポジトリは、これに基づいてソースコードを解析した結果を直接解釈実行することが可能な仮想機械を作成できるなど、これまでのモジュールや関数単位のリポジトリに比べて、十分に細かい粒度を持っている。このことは、Sapidが動的解析に有効であることを示している。しかし、Sapidを動的解析に応用した例は多くない。さらに、SapidはCASEツールをのプラットホームになることを目的に設計されている。CASEツールを開発するときにしばしば必要になる構文解析器や字句解析器などの汎用的なモジュールを提供することで、開発の省力化やCASEツール間のデータ統合なども期待できる。

1.2 目的

以上の背景をふまえて、本研究では目的をSapidを用いてC言語のなるべくフルセットに対応できるダイナミックスライサーの実装を行うことに置く。そして、動的解析およびダイナミックスライシングを困難にしている要因を明確にし、それに対する回答を探る。また、その過程で、Sapidの有効性、特に動的解析に対する有効性を示すとともに、その問題点を探っていく。

1.3 アプローチ

ダイナミックスライシングの最も一般的なアルゴリズムは、プログラムの実行系列をスライシング基準から逆向きに制御依存関係とデータ依存関係にしたがってたどっていくものである。まず、プログラムに入力を与えて実行し、その実行系列を抽出する。次に、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たどっていくのである。今回の実装では、これに加えて命令同一関係も追跡するKorelのダイナミックスライシングのアルゴリズム[BJ90]を採用する。これは、ループから確実に抜けられるようにするためである。

これをSapidを用いて実装する。Sapidには動的解析を支援するためのライブラリSIP、SIP2が用意されている。また、これらを用いて実装されたC言語の簡易インタプリタも提

供されている。そこで、このインタプリタを改造して実行系列を抽出することにする。プログラムが 1 ステップ進む度に、そのステップで実行されて命令、参照された変数、定義された変数、影響を受ける分岐命令をリストにして構造体に格納していくことにする。

ポインタについての追跡は Points-to 集合 [SH97b] を抽出することによって行う。各ポインタがそれぞれの実行時点で参照している変数をたどり、定義や参照された変数として構造体に格納していく。実行系列が抽出できた後は、スライシング基準からデータ依存関係、制御依存関係、命令同一関係を構造体にアクセスしながらたどっていく。ポインタでも定義、参照関係が明らかになれば基本的には他の変数と同様な操作でスライスを得ることができる。

実装には Sapid の提供するライブラリを用いるので、コードの量は極端に少なくなることが予想できる。構文解析、字句解析はもちろん実行環境も Sapid の提供する簡易インタプリタを用いるので、純粋にダイナミックスライシングのアルゴリズムの部分の実装だけで済むはずである。

今回の実装に用いる Sapid の簡易インタプリタである sint は C 言語のすべての機能に対応しているわけではない。ライブラリ関数やシステムコールには一部しか対応していない。また、配列の定義の仕方に制約があり、ポインタの演算にも対応していない。したがって、実装するスライサもこれらに対応しないものとする。しかし、ライブラリ関数については一部実装されており、他のライブラリ関数も同様のアプローチで実装できると考えることができる。システムコールも成功を仮定するなどすれば一部については同様に実装できると考えられる。

1.4 特色

Sapid の細粒度という性質はは動的解析に対し有効なはずである。プログラムを実行するのに必要な情報は、構文に基づいてすべて保存されているからである。けれども、これを本格的に動的解析に応用した例はまだない。また、実用的な言語に完全に対応したダイナミックスライサの実装は多くない。さらに、Sapid という統一したリポジトリを利用することにより、他のツール群との統一のとれた連携も考えることが可能である。

1.5 本稿の構成

本論文はまず、1 章で研究の背景と目的、特色を述べる。次に 2 章でスライシングの概要と、問題点をあげる。スタティックスライシングとダイナミックスライシングの相違と

それぞれの特色、代表的なアルゴリズムや、スライシングの応用例、問題点などについて述べる。3章ではSapidの概要と本研究での位置づけを述べる。まず、Sapidの細粒度を特徴づけるソフトウエアモデルや、提供されるAPIやツールについて述べる。4章でダイナミックスライサの設計と実装を説明する。実装の問題点について述べる。Sapidのどの部分をどう利用するのか、それがどのような利益をもたらすのかについて述べる。実装の方針と設計、採用した具体的なアルゴリズム、について述べる。最後に5章で本研究の成果と今後の課題について述べる。

第 2 章

Slicing

2.1 Introduction

プログラムスライシング [Wei84] は 1982 年にメリーランド大学の Mark Weiser 博士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。

2.2 概要

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。いか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。

ほかにも、スライシングは二種類に分類できる。スタティックスライシングとダイナミックスライシングである。この二つの違いは、入力を限定するかしないかである。限定しない方をスタティックスライシングといい、限定する方をダイナミック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。いかにそれぞれの特徴を簡単に述べる。

- スタティックスライシング
  - 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
  - 実装はダイナミックスライサに比べて容易
- 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
  - 実装はスタティックスライサに比べて困難

2.3 Static Slicing

2.3.1 定義

ある変数に関してはもと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スタティックスライスという。　スタティック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したがってたどることによって得ることができる。

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。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。

配列の要素の最小値と最大値、合計を求めるプログラムのフローグラフは図 2.1のようになる。

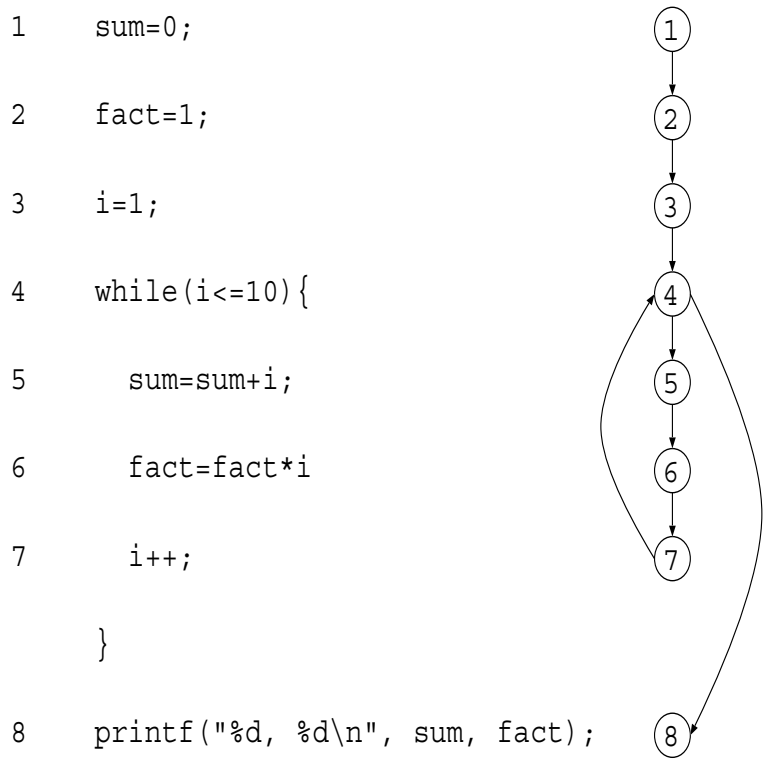

図 2.1: フローグラフの例

また、スタティックスライシングではデータ依存関係と制御依存関係は次のように定義される。

- 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。ここでいう「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。ここでいう「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。

2.3.2 方法

スタティック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$C=(u,V)$のように書く。スライシング基準$C=(u,V)$に関するスタティックスライス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. $SS=\bigcup_{i=0}^{\infty}A^i\bigcup\{u\}$

これをもとに図2.1のプログラムをスライシング基準$C=(13,"min")$について求めると図2.2となる。基本的なスライシングの方法は以上である。

2.3.3 まとめ

スタティックスライシングは、実行環境を用意する必要がないぶん実装はダイナミックスライサに比べれば容易である。しかし、ポインタや配列、構造体を正確に取り扱う方法がまだ確立されておらず、配列の要素をすべて一つの変数として取り扱うという方法が取られることが多い。したがって解析が雑になり、正確なスライスは求められない。フローグラフをすべてたどることになるので、探索範囲は広くなってしまう。

```
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 2.2: スタティックスライスの例

2.4 Dynamic Slicing

2.4.1 定義

ある入力を与えてプログラムを実行したとき、ある変数に関してはもと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ダイナミックスライスという。スタティックスライスでは、任意の入力を与えて実行した場合を対象としているのでフローグラフ上のすべての実行系列を考える。しかし、ダイナミックスライスは特定の入力を与えて実行した場合のみを対象としているため特定の実行系列しか考えない。

入力を与えてプログラムを実行した場合、実行された命令の列を実行系列という。また、実行系列上において、n番目に実行された命令を指して、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)に入力 n=3,A=(2,1,3) を与えて実行したときの実行系列は表 2.1のようになる。

実行系列	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: 実行系列の例

ダイナミックスライスも基本的にはデータ依存関係と制御依存関係をたどることによっ

```
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: プログラムの例

て得ることができる。スタティックスライシングでは依存関係はフローグラフについて考えたが、ダイナミックスライシングでは実行系列について考える。スタティックスライシングとダイナミックスライシングにおける依存関係を区別するときは、前者を静的依存関係、後者を動的依存関係という。ダイナミックスライシングでは動的依存関係と静的依存関係は次のように定義される。

- データ依存関係，$DU(p,q)$
  実行時点 qで参照される変数 wが存在して、実行時点 pが実行時点 qより前で最後に wを定義しているとき、「p から 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ことは、pで定義した変数を qで参照したことを意味している。

- 制御依存関係，$TC(p,q)$
  実行時点 qで実行された命令と静的制御依存関係がある命令の中で、q以前で最後に実行された命令を実行しているのが、実行時点 pのとき、「pから 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ことは、命令 qが実行されたことが命令 pの実行結果に依存していたことを意味する。

2.4.2 方法

ダイナミックスライシングではスライシング基準はプログラムに与えられた入力 x、プログラムに xを入力したとき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で与えられる。これを $C=(x,r,V)$のように書く。スライシング基準 $C=(x,r,V)$に関するダイナミックスライス 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 $TC(r,v)\}$
   ただし、$Def(r,v)$は実行時点 rより前で変数 wに最後に値を定義した実行時点を表す。

2. $i\geq 1$に対して、$A^1=\{$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 $TC(p,q)\}$

3. $DS=\{Ins(p)|p\in\bigcup_{i=0}^{\infty}A^i\bigcup\{r\}$

これをもとに図2.3のプログラムをスライシング基準 $C=(\{n=3,A=(2,1,3)\},19,sum)$について求めると図2.4となる。

スライシング基準が $C=(\{n=3,A=(2,1,3)\},19,sum)$のときはこれで正しくダイナミックスライスが求まっている。では、スライシング基準 $C=(\{n=3,A=(2,1,3)\},19,sum)$についてのダイナミックスライスをもとめてみる。すると図2.5となる。

しかし、これには12行目のiのインクリメントが含まれていない。これは、正しいダイナミックスライスではない。ループ文で、一回目のループでのみしか実行されない命令から依存関係をたどるとこういうことが起きてしまう。そこで、命令同一関係を次のように定義し導入する。

- 命令同一関係 $IR(p,q)$
  実行時点 pと実行時点 qにたいして、$Ins(p)=Ins(q)$のとき、「pと qの間に命令同一関係がある」という。

これにともない、ダイナミックスライスの求め方を変更する。

2. $i\geq 1$に対して、$A^1=\{$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 $TC(p,q)$もしは $IR(p,q)\}$

このアルゴリズムにしたがって、もう一度同じスライシング基準でダイナミックスライスを求めてみると図2.6となる。

今度は、iのインクリメントも含まれ正しくダイナミックスライスが得られている(図を見易くするために実行系列で8、15の間の命令同一関係は記述していない)。この命令同一関係も導入したダイナミックスライシングのアルゴリズムがKorelのアルゴリズムである。

基本的なダイナミックスライシングの手法は以上である。

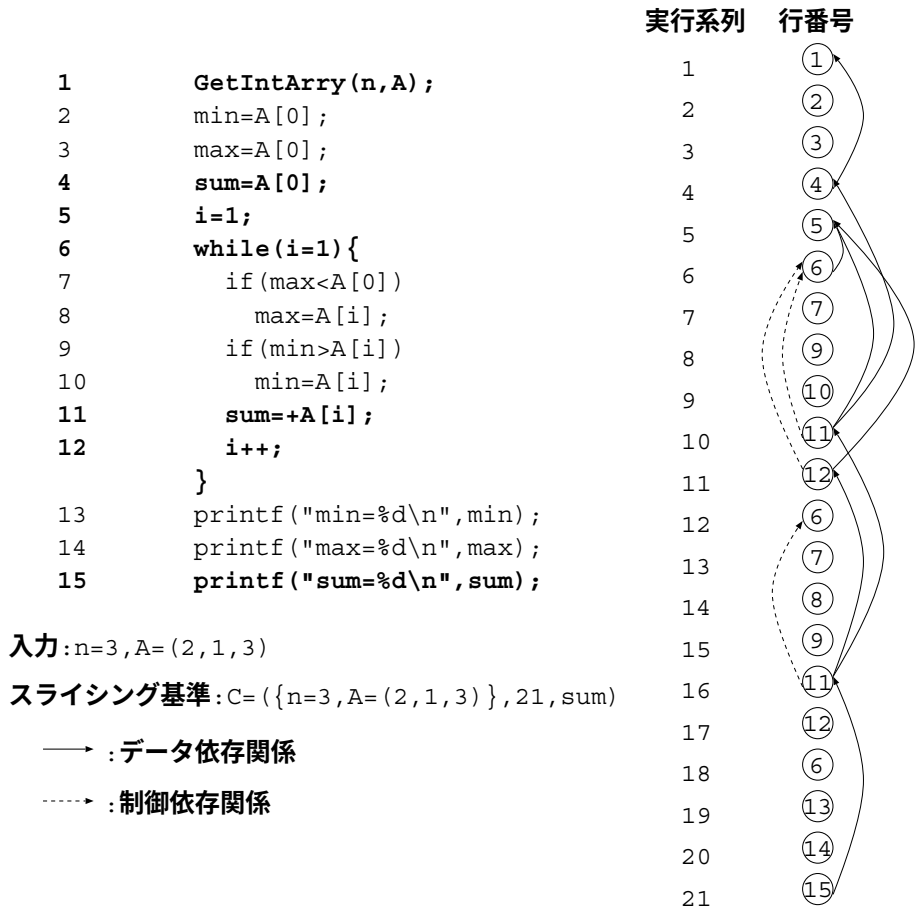

図 2.4: ダイナミックスライスの例 1

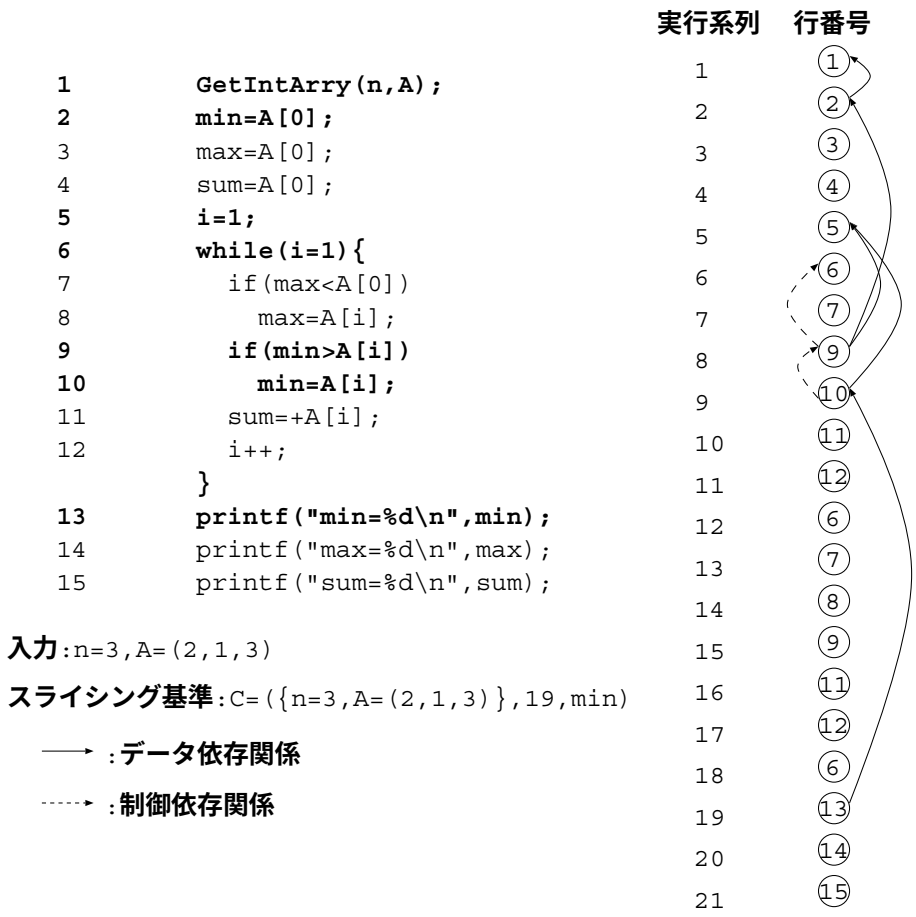

図 2.5: ダイナミックスライスの例 2

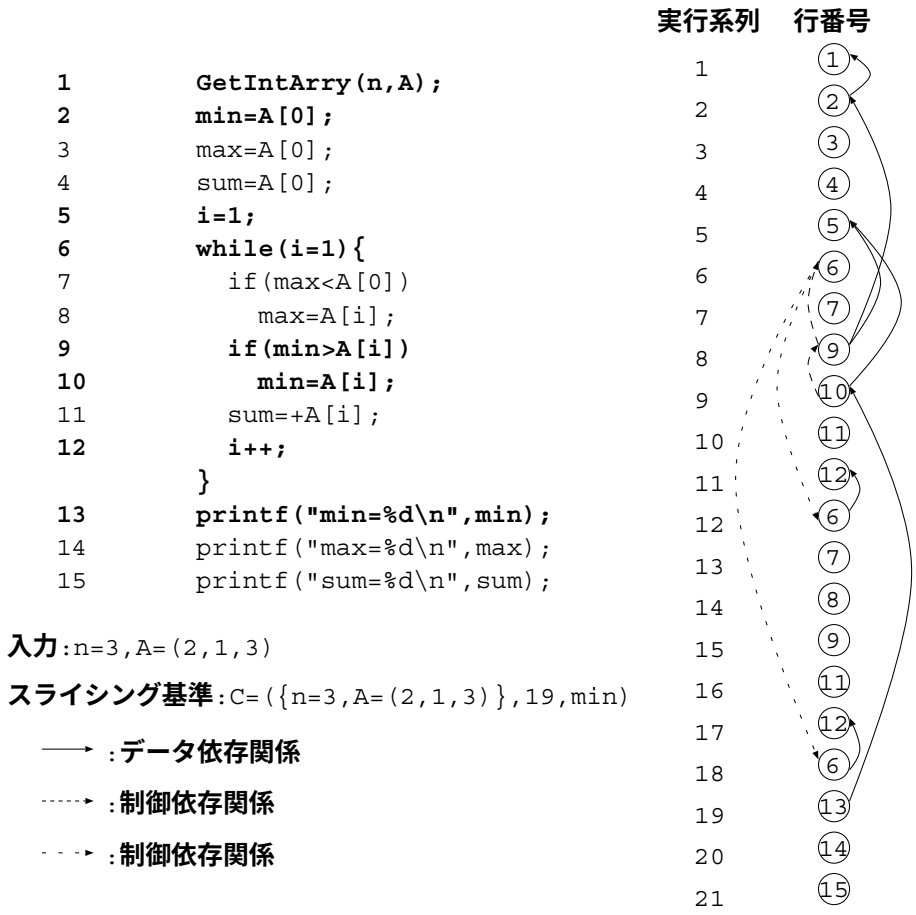

図 2.6: ダイナミックスライスの例 3

2.4.3 まとめ

ダイナミックスライシングはアルゴリズムは単純であるが、実行環境を用意しなければ解析が行えないので、実装は容易ではない。しかし、入力を固定することにより実行系列が決まるので、探索範囲を著しく限定でき、正確なスライスを求めやすい。実行状態をすべて保存してから解析を行うことになるのでメモリの制約が生じる場合がある。また、一般的に動的な情報を用いることが可能なのでスライシングをさらに正確にできる可能性がある。

2.5 応用

スライシングは様々な技術に応用されている。目的にあわせて、適切なスライスを求めることが要求されている。

2.5.1 デバッグ

スライシングはもともとデバッグを目的に考案された。予期しない値を持った変数に関してスライシングをすることにより、その変数に予期しない値を代入する原因になった命令を抽出することができる。スタティックスライシングを用いれば、プログラム内からエラーを引き起こした可能性のある部分を抽出できる。また、ダイナミックスライシングを用いればエラーを引き起こした可能性のある命令を実行した実行時点をシステムが解析しバックトラックをすることにより、その実行時点におけるプログラム状態を自動的に再現できる。

2.5.2 テスト

プログラムの品質を向上させるためには、十分なテストを行うことが必要である。テスト項目の作成方法には、プログラムの機能仕様をもとにして入力変数の値を適切に組み合わせてテスト項目を作成する機能テストと、プログラムの構造をもとにしてプログラムのなるべく多くの部分をテストできるようにテスト項目を作成する構造テストの手法がある。構造テスト手法には、プログラム内のすべての分岐をテストするブランチテスト、すべての実行パスをテストするパステスト、変数に値が定義されてから参照されるまでのパスをテストするデータフローテストなどがある。そして、このテストデータの自動抽出にスライシングが有効である。

2.5.3 再利用

プログラムは数千行、数万行といった規模のものも珍しくない。しかしその一部の記述は以前に書かれたプログラムと同じものであることがある。そのコードを再利用できれば、プログラミングの手間を省くことができる。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の効率を上げることができる。プログラムスライシングを使うことによって欲しい機能を実現している部分を正確に取り出せることが期待できる。

2.5.4 統合

プログラムの再利用は頻繁に行われている。いま別のプログラムから取り出したある機能を実現している部分を、別なプログラムに統合することを考える。統合したプログラムでは統合した機能が他の部分に影響を与え予期しない動作をしないこと、統合した機能が他の部分の影響を受け予期しない動作をしないことが要求される。統合したプログラムをスライスすることにより、統合した機能この独立性を確認することができる。

第 3 章

Sapid

3.1 Introduction

Sapid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CASEツール・プラットホームである。

ソフトウエア工学では、ソフトウエアの生産性を向上させるためにリポジトリによるデータ統合などの様々な標準化作業が行われてきている。しかし、これまでの統合化は比較的粒度の荒い関数やモジュールを対象としており、その構造や構成などのデータは取り扱われていない。ところが、これらのデータの中にも実装や保守、再利用などにおいて重要な役割を果たすものがあり、それを無視することはできない。

また、ソフトウエアの生産性を向上させるために現在、様々なツールが利用されているが、個々のツールは独立して存在しており、ソフトウエアのライフサイクルの特定の局面だけしか支援することができない。特定の局面だけでなく、要求分析から保守・運用までのすべての局面を統一的に支援することが求められている。

一方で、CASEツールの研究やその試作においてプログラムの解析器等の実装が必要になることがあるが、それを個別のCASEツールが独自に持つのは無駄である。解析器の実装はCASEツールの実装において本質的な問題ではないにも関わらず、多くの労力が必要だからである。CASEツールに共通して用いられる機能を提供するようなCASEツールプラットホームがあれば、研究者はCASEツールの本質的な機能の実現に専念することができる。

そこで提案されたのがSapid(Sophisticated Application Programming Interface for software Development)である。モジュールよりも小さい成果物を対象とするリポジトリ、細粒度リポジトリを提案し、そのリポジトリに基づいた様々なAPI群やツール類を提供するこ

とで、上記の問題に対する回答を出している。

Sapidの現在の最新のバージョンはver.3.587であるがこの論文を書き出したときはver.3.243であり、この論文はver.3.243に基づいている。

3.2 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エ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、粒度の荒いもを細かい粒度にするのは不可能であるが、細かい粒度を荒ら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、ソフトウエ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、ツール群の三つの部分から成る。

- SDB
  Sapidの基盤となるソフトウエアデータベースをSapidではSDBと呼んでいる。I-modelをはじめとしてP-model, Area-modelなどのモデルに基づいて構成されたSapidの根幹となるデータベースである。

- API群
  SDBにアクセスするための関数と、CASEツールを作成するために用意された基本的な関数、ソースコードに対する変更を行う関数などからなるAPI群。データベースから得られた情報を解析などする基本的な関数の集まり。これらをSapidではAR(Access Routin)と呼んでいる

- ツール群
  Sapidを用いてCASEツールを実装するときに必要になるツール群。ソフトウエア

データベースをブラウズするツールなどが含まれ、Sapidの APIを用いて実装されている。

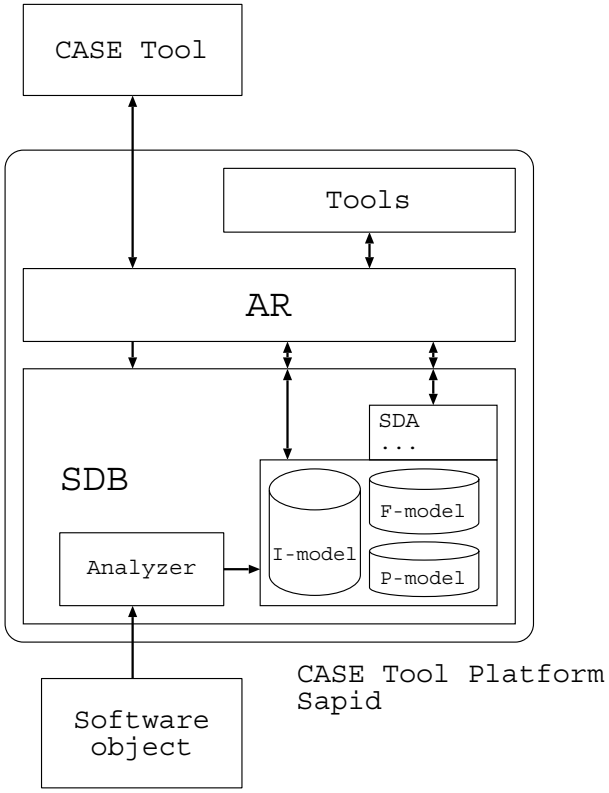

図 3.1: Sapidの構成

3.2.1 SDB

SDBは C言語のソースコードを解析した結果を格納するソフトウエアデータベースである。ソフトウエアの構成要素にはソースコード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。また、C言語以外の言語にも対応していない。

Sapidでは C言語のソースコードをいくつか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 ER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。

- I-model

  I-model[有賀]はC言語の構文を基にソフトウエアを捉えたモデルである。前処理されたC言語のソースコードに対するソフトウエアモデルで、前処理後のソースコードをprogram, file, block, expressionなどからなる12の実体と、その構成関係、定義参照関係などを表す29の関連からなるERモデルとして捉えている。前処理後のC言語のソースコードに対するモデルであるのは、前処理に関する部分はC言語の構文規則従っていないためである。そのため、P-modelが用意されており前処理以前のソースコードとの対応を取ることができるようになっている。C言語の構文規則をもとに実体と関連を定義しているが、ソフトウエアの開発に必要か不必要かに応じて取捨選択して作られている。I-modelのその構成関係をOMTモデルのオブジェクト図の記法で表すと図3.2のようになる。

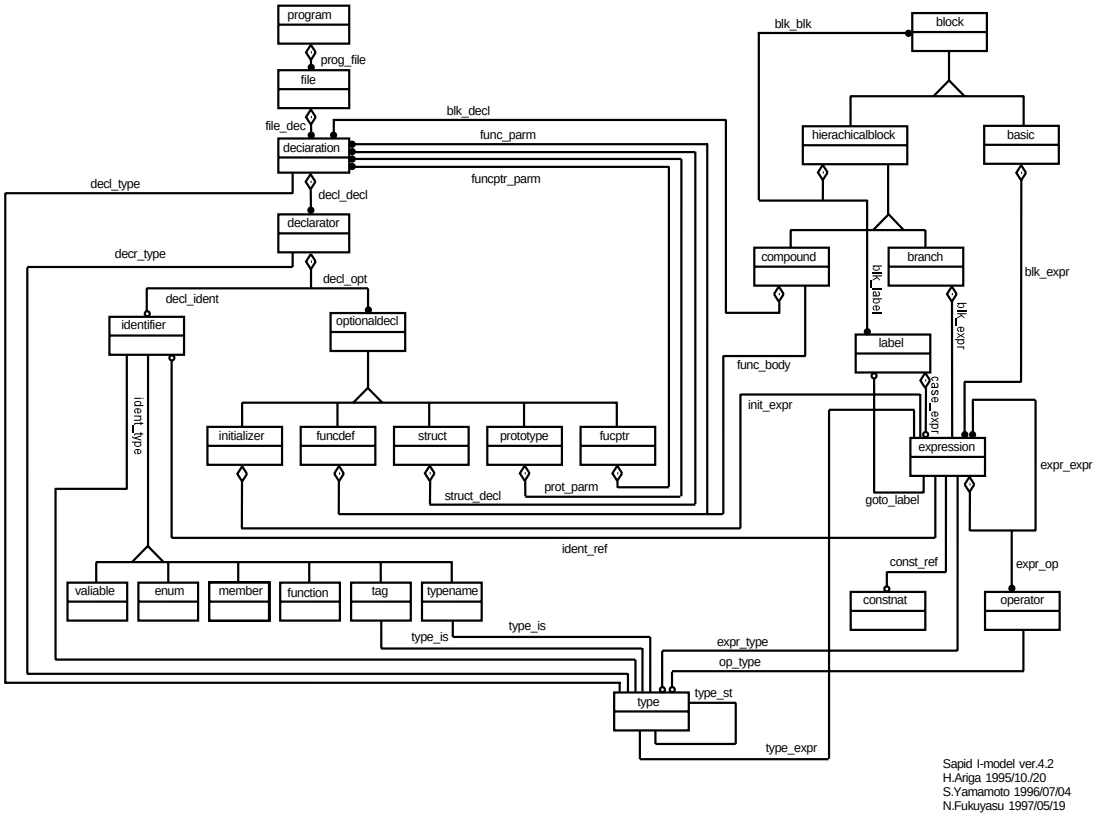

図 3.2: I-modelのオブジェクト図

各オブジェクト(個々の実体および関連)はいくつかの属性を持ち、オブジェクトの性質などを表している。主な実体と関連の主な属性には図3.1のようなものがある。

実体の主な属性	
id	実体の識別番号
name	名前
length	byte単位の長さ
sort	種類

関連の主な属性	
id	関連の識別番号
obj1_id	一方のオブジェクト obj1の Id
obj2_id	他方のオブジェクト obj2の Id
offset	ファイル先頭からの位置

表 3.1: 実体と関連の主な属性

例として図3.2に識別子を表す実体の属性を示す。

- P-model, F-model
  この二つのモデルは前処理命令に対するソフトウエアモデルである。I-modelを利用してソースコードを解析した場合利用者は前処理後の情報しか得ることができない。しかし、実際に利用者が見ることになるソースコードは前処理前のソースコードであるので、その整合性を取るモデルが必要になる。

  P-modelは前処理命令を10のクラスと15の関連としてモデル化している。前処理命令を変数や関数呼び出しのように解釈し、I-modelのように解析を行い、SDBの中に、前処理情報のデーターベースとしてPIDBを構築する。。前処理後のソースコードから前処理前のソースコードを得る手法を提供しようとしている。

  利用者がソースコードに変更を加える場合、前処理前のソースコードに変更を加えることを望むが、I-modelに基づいて解析を行ったあとではそれは不可能である。ソースコードを前処理前の形に戻す機能もSapidでは提供されているが、不完全で、ソースコードに変更を加えた場合はさらに不完全なものになる。F-modelでは前処理前のソースコードに対する変更を目的に設計されている。

Sapidではこれら以外にも、ソースコードにおける文をI-modelとは異なる形で解釈したStatementModel、メモリ上の領域と値に関する情報モデルであるArea-modelなどのモデ

クラス名	属性名	属性の型	属性の説明	
	identifire			
	id	Integer	objectの識別番号	
	name	String	objectの識別番号	
			マクロ定義	種類
			ID_VARIABLE	variable
			ID_FUNCTION	function
			ID_MEMBER	member
			ID_ENUMERATOR	enumerator
			ID_TYPENAME	typename
			ID_TAG	struct tag
	length	Integer	objectの識別番号	

表 3.2: 識別子の属性

ルがある。

3.2.2 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
  もっとも基本的なAPIがAR4である。このAPIにはI-modelとP-modelのSDBに関するAPIである。AR4はAccess Routinesの略で、文字通り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なめ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はSapid Dependency Analyzerの略で、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。

3.2.3 ツール類

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、ソースコードの理解支援ツール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。

第 4章

ダイナミックスライサの実装

4.1 Introduction

この章ではこれまでの章で述べてきたことをもとにいかにしてダイナミックスライサを実装するか、その方針について述べる。また、実装するときの問題点についても述べる。

Sapidが動的解析に対してもいかに有効かをまず述べる。Sapidの有効性を持ってしても、解決しがたい問題としてのポインタ解析を説明し、その問題を解決するためのC言語のサブセットを説明する。それに基づいてスライサの設計の方針を述べる。採用したKorelのアルゴリズムがどのように実装されるか、ポインタの定義参照関係をどう抽出するかを述べる。最後に今回実装した部分とその設計の概略図について説明する。

4.2 Sapidの有効性

Sapidはダイナミックスライサの実装を有効にサポートすることができる。3章で述べたように、ダイナミックスライシングでは実行系列を抽出することが重要である。しかし、実行系列を抽出するには入力を与えてプログラムを走らせる必要があり、これは容易ではない。何らかの実行環境が必要で、実用的なプログラミング言語においては実装が巨大になってしまうからである。しかし、細粒度リポジトリであるSapidを利用すれば、ダイナミックスライシングに必要な情報をソフトウエアデータベースか取り出しながら、C言語プログラムをステップ実行することができるので、実装の手間を極端に減らすことができるはずである。ある実行時点で、構文上でどの変数が参照されたのか、どの変数が定義されたのかを調べられる。これがわかれば、基本的にデータ依存関係は抽出できる。また、その命令が属するブロックを調べることで、制御依存関係もわかるのである。

Sapidには動的解析を目的としたSIP、SIP2というパッケージが用意されている。そして、これを利用して実装されたC言語のインタプリタであるsintも用意されている。sintは高々104行の短いプログラムであるが、C言語の実行環境を完璧ではないものの実用的に用意している。これを改造することにより実行系列と、各実行時点で定義参照されている変数を容易に得ることができるはずである。したがって、実装しなければならない部分は主にダイナミックスライサの本質的なアルゴリズムの部分だけになるはずである。

しかし、Sapidにも問題点がないわけではない。SapidのC言語インタプリタであるSintがサポートしていない機能がいくつかあるからである。

- 構造体のメンバにアクセスしたとき正しい値が得られないことがあること
- 対応しているライブラリ関数が極端に少ないこと
- 配列にアクセスすると`Bus Error`を起こすことがあること

などが、上げられる。しかし、これらの問題は解決できない問題ではなくSapidやSintの開発が進めば解消することが期待できる。

また、Sapidでは、SDAによって依存解析を行う仕組みを用意している。しかし、今回の実装では使用を見送った。SDAは基本的に静的な解析をするものであり、

- 関数に跨る依存関係が解析できない
- 大域変数を大域変数として扱えない
  関数内で大域変数を定義しないと、その大域変数の値にアクセスできない
- 関数の戻り値を扱えない
- 配列の要素をすべて一つの変数として扱ってしまう
  `a[2]=p;`という命令があったとき、`a[0]`, `a[1]`, `a[2]`,..., `a[n]`すべてが`p`によって定義されたと判断してしまう

などの問題があったからである。

4.3 ポインタ解析の難しさ

スライシングを求めるときに困難な問題が二つある。一つは、ライブラリ関数とシステムコールで、二つ目がポインタ解析である。

ライブラリ関数やシステムコールはソースコードがプログラムに含まれていない。したがって、データ依存関係がライブラリ関数やシステムコールによって切られたり、変更されてもそれを知るすべがない。

C言語では正確なポインタ解析は困難とされている[HS, SH97a]。これには主に二つの理由がある。一つ目の理由は、C言語では他の言語に比べて自由なポインタの操作を認めていることである。二つ目の理由は、一つ目の理由を利用して処理系や、ハードウエアに依存したコードを書けることである。

C言語ではポインタを自由に操作して、何を指しているのかわからないポインタを合法的かつ簡単に作ることができる。次の例はすべて合法的なポインタ操作であるが、操作の結果ポインタは何を指しているかわからなくなる。

```
int *p;                    int *p;
p=(int *)135;              int a;
*p=3;                      p=&a;
                           p++;
                           *p=12;
```

図 4.1: 意味のないポインタの例

これらはコンパイル時にエラーを出さないだけでなく、実行時エラーを起こさない可能性すらある。ポインタを使って値を参照する場合、ポインタが正しいアドレス、つまり変数の宣言もしくは`malloc()`等で確保したメモリの、それぞれの変数の先頭アドレスを指していることが要求されるが、それをC言語は保証しないし、また正しくないアドレスを指すことを禁止もしない。

C言語では言語処理系や、ハードウエアに依存したコードを書くことができる。例えば、構造体の中のメンバのメモリ上の配置は処理系依存である。

上の例は`e`で`A.d`をアクセスできるがどうかは処理系によって決まる。`int`の具体的なサイズはANSIのC言語の規格では定められていないし、メモリ上の配置も`A`のアドレスと`A.a`のアドレスが等しいこと以外は定めていない。多くの処理系で上のプログラムは`A.d`を`e`を用いてアクセスできるが、これはたまたま動いているにすぎない。仮にアクセスできていたとしても`c`と`d`の間にメモリアクセスの便宜を図るためにダミーのデータが入っているのでたまたまアクセスできているだけである。

```
struct{
    char    a;
    char    b;
    char    c;
    int     d;
}A;
int     e;
A.a=41;
A.b=42;
A.c=43;
A.d=44;
e=*((int *)&A+1);
```

図 4.2: 実装依存なソースコード

C言語ではシステムに密着したコードを書くことができる。このことがC言語の、いわゆる高級言語でありながらシステムプログラミングにも対応できるという柔軟性をもたらしている。しかし、その代償として解析が困難なバグと移植性の低さを産み出す土壌を作り出している。

4.4 C言語のサブセット

前章で述べたように、C言語はシステムに密着したコードを書くことができる。これは致命的なバグと移植性の低さを産み出しやすくしている。ゆえにそのようなコードは好ましくないとされている[WP99]。そこで、今回の実装ではC言語に小さな制約を設けることにする。その制約とは次のようなものである。

- 暗黙明示を問わずキャストは認めない。ただし`malloc()`を目的のポインタ型にキャストすることは認める。
- `malloc()`と`free()`以外のライブラリ関数とシステムコールは認めない。

キャストを禁止することにより、ポインタ変数は`malloc()`で確保したアドレスと&演算子で得られたアドレス、この二つからポインタ演算をして得られたアドレスしか指さないことになる。これと、副作用を持つ演算を解析するルーチンを用いれば、ポインタ変数が正しくないアドレスを指しているかどうかを調べることができるようになる。

副作用を持つ演算が式の中に現れると式の評価の途中で、ポインタが参照しているアドレスが変わる可能性が生じる。しかし、式を等価な複数の式に分割することでアドレスの変化を追跡することができる。この機能をSapidは提供しないが、Sapidを使えば容易に実現することができる。

また、ライブラリ関数とシステムコールを禁止することにより、解析をすべてソースコードに基づいて行うことが可能になり、データ依存関係を追跡できるようになる。しかし、`malloc()`と`free()`を禁止してしまうと、動的なメモリ割り当てが不可能になり、動的解析の意味が半減してしまうので`malloc()`は例外とした。

4.5 設計

今回の実装ではダイナミックスライシングのもっとも一般的な実現方法である、Korelのスライシングアルゴリズム[BJ88, BJ90]を用いる。このアルゴリズムはデータ依存関係と制御依存関係、および命令同一関係をたどることによりスライスを得るものである。

データ依存関係を抽出するには、各実行時点で参照されている変数と定義されている変数を保存しておく必要がある。インタプリタによってプログラムをステップ実行しながら、参照された変数のリストと定義された変数のリストを作っていく。このリストは実行時点ごとに作られる。作られた二つのリストを構造体に格納する。構造体のメンバには、他に実行時点、命令文のID、実行した命令が含まれるブロックがある。この構造体も実行系列に従ってリストになっている。

これにより実行系列に関するデータベースができあがる。これを第4.8章で述べるPoints-to集合の抽出と組み合わせて最終的な実行系列に関するデータベースができあがる。

参照された変数のリストと定義された変数のリストから、データ依存関係がわかる。命令文のIDから命令同一関係がわかる。命令が含まれるブロックの属性から制御依存関係がわかる。つまり、ダイナミックスライシングに必要な情報は実行系列データベースにすべて格納されていることになる。

プログラムの実行が終わったあとで、実行系列を表示する。次に、実行系列からスライシング基準を選択してもらう。スライシング基準が妥当かどうかを判断する。そのあと、スライシング基準から各依存関係と命令同一関係をたどっていく。

```
    ProgramStep = 1
    Program 実行
1:  NewStructure を確保
    ProgramStep を NewStructure に追加
    StatementId を NewStructure に追加
    BlockId を NewStructure に追加
    DefinedVariablelist 取得
    DefinedVariablelist を NewStructure に追加
    ReferedVariablelist 取得
    ReferedVariablelist を NewStructure に追加
    NewStructure を PrevStructure にリンク
    PrevStructure = NewStructure
    Program 一ステップ実行
    ++ProgramStep
    1にジャンプ
```

図 4.3: 実行系列のデータベースを作りかた

4.5.1 Sapidの利用

今回は、インタプリタとしてSIP2を用いて実装されているsintを改造することにした。SIPを用いて実装されているsipよりも、高機能でソースコードが理解しやすかったからである。

Sapidには依存関係を追跡するためのツールSDA3およびSDA4が用意されているが、今回は使用しなかった。理由には、関数に跨る依存関係を追跡できないこと、大域変数が扱えないことなどがある。

各命令文の中から参照、定義される変数を取り出すのにはAR4を用いた。AR4にはSDBにアクセスするための基本的な関数しか用意されていないが、その分制約が少ないからである。

4.6 アルゴリズム

上でダイナミックスライサの処理の大まかな流れを述べた。ここでは、参照定義された変数のリストの生成方、各依存関係と同一関係のたどり方について述べる

4.6.1 参照定義された変数リスト

sintによる実行の過程で得られるのは実行している命令文のIDである。まず、その命令文から参照定義されている変数と、属するブロックの制御文のIDを調べなければならない。

- 定義参照されている変数のID
  AR4の間連取得関数をもちいて、まず関連expr_exprを関連ident_refが現れるまでたどる。ident_refが現れたら、ident_refをたどり変数のIDをえる。さらに変数の属性が左辺値か右辺値か、またはその両方かを調べる。左辺値ならば定義されている変数のリストに、右辺値ならば参照されている変数のリストに、両方ならば二つのリストに、変数のIDを加える。expr_exprを闇雲にたどると探索が循環してしまうので、次の探索に移る前に現在のexpressionのIDを記憶しておき、次の探索ではそのIDに関する探索結果は無視する。

- ブロックの制御文のID
  AR4の関連取得関数をもちいて、まずblk_exprをたどる。次に関連の属性にcompoundが現れるまでblk_blkをたどる。compoundが現れたらもう一回だけblk_blkをたどる。そのとき、ブロックの属性が分岐ブロックだったら、blk_exprをたどり、条件式等(for

ループだと3個ある)のIDをすべて得る。もし、そのブロックの属性が分岐ブロックでないのなら、リストのポインタはNULLのままにしておく。blk_blkもexpr_exprと同様に探索が循環しないようにする必要がある。

4.6.2 依存関係と同一関係のたどり型

スライシングは再帰的に定義されているので探索も再帰的に行った。スライシング基準からいずれかの関係をたどることで、別の実行時点のある変数が依存関係を持つことがわかったとする。この、実行時点と変数をスライシング基準に持つ新たなスライスAを考える。最初のスライシング基準から得られたスライスをBとすると、AはBに含まれるはずである。したがって、関係を一つたどったら、たどった先を新たなスライシング基準と見なすことにする。データ依存関係と制御依存関係、命令同一関係は次のようにして追跡を行う。

- データ依存関係のたどり方
  まず、スライシング基準になっている変数に注目する。その変数を定義している命令を実行系列を逆順にたどりながら探す。一つ見つかったら探索は終了する。その命令で定義した値しか基準に到達しないからである。実行系列を先頭までたどっても見つからなかった場合はその変数が定義されずに使用されていたことを意味する。この場合はデータ依存関係の追跡は終わったことになる。最後に、見つかった命令で参照されている変数すべてを新たなスライシング基準と見なす。もし、見つかった命令で参照されている変数がなければ、データ依存関係の追跡は終わったことになる。

- 制御依存関係のたどり方
  まず、スライシング基準になっている文に注目する。その文が属するブロックの制御文のIDのリストを見て、もしそれが空ならばその文は他の文と制御依存関係はない。空でなければ、制御文のIDリストにある文と制御依存関係があることになるので、それぞれの文のIDを新たなスライシング基準と見なす。

- 命令同一関係のたどり方
  まず、スライシング基準になっている命令に注目する。実行系列の中からその命令と同じIDを持っている命令を探す。実行系列のなかすべてを探索しなければならない。もし見つかったなら、そのすべてを新たなスライシング基準と見なす。見つからなければ、命令依存関係の追跡は終了したことになる。

この三つをスライシング基準にすべて適用し、かつそれを再帰的に繰り返すことによりダイナミックスライスを得ることができる。

4.7 Points-to集合

ポインターを追跡するときに広く使われている手法にPoints-to集合[SH97b]を求めるものがある。Points-to集合とはあるポインタ変数がさしている可能性のある変数の集合のことである。Points-to集合の大きさは、そのPoints-to集合を得るのに用いたポインタ解析の正確さを表している。集合が小さい方が正確なポインタ解析であることをあらわしている。

```
int a[3];
int *p;
int *q

p=&a[0];
q=p;
```

図 4.4: Points-to集合

例えば上のプログラムで、`*q=a[0]`である。したがって、配列を正しく解析していればPoints-to集合は{`a[0]`}となる。しかし、配列の添字を解析しないような解析器であると{`a[0], a[1], a[2]`}となってしまう。

4.8 Points-to集合の抽出

ここではポインタの定義、参照をどのように導くかを説明する。また、その関係からどの様にデータ依存関係をたどるかを示す。

ポインタの定義には三種類しかない。

- pがポインタ: `p=&q`
- pがポインタ、Aは何らかの型: `p=(A *)malloc(SIZE)`
- pがポインタ、nは int: `p=p+n, p++, p--`

すなわち、ポインタ値は&演算子とmalloc()関数、ポインタ演算でしか生成できない。第4.4章の仮定からこのことがいえる。この場合、ポインタpが左辺値として現れるので、第4.6章と同じアルゴリズムですむ。

ポインタの参照は複雑である。変数と、その変数を参照しているポインタの間にはデータ依存関係があるからである。ある変数を参照しているポインタが存在するかどうかは、ポインタの参照するアドレスをすべて調べ、変数のアドレスと同じかを確認するしかない。なぜならポインタ値は動的に決まるからである。そこで、すべての実行時点の、すべての変数に対して、次の情報が得られると仮定する。

- 変数のアドレス
- 変数がポインタなら参照しているアドレス

sintではこの情報は得られないが、実行環境が正しくポインタを扱っているならば、その情報は各々の実行時点で得られるはずである。各々の実行時点でその情報が得られ、その情報を取り出せるものとする。このもとで、ある実行時点で参照されている変数を考える。

- ある実行時点に右辺値として現れる、すべてのポインタ変数pに対して次のことを行う。

  1. 変数pが参照しているポインタをすべてたどる。ついている*の数だけ、参照しているアドレスをたどっていく。
  2. たどったアドレスを持つ変数を探す。あればその変数は参照されている。

  これはポインタが参照している変数もその実行時点で参照されている変数だからである。この追跡は上の二つの仮定より可能である。

  例)a=(**p), b=(*p)なら、pはaとbも参照していると考える。c=pのときは、参照しているとは考えない。

これで、ある実行時点でポインタを通して参照されている変数がすべてわかったことになる。次に、どの実行時点でポインタを通して変数に値が定義されているかどうかを調べるアルゴリズムを考える。これにより、ポインタがあるときでもデータ依存関係がたどれるようになる。

- ある実行時点で参照されているすべての変数に対して、次のことを行う。いま、対象としている変数を仮にaとする。

1. aを参照しているポインタがあるかどうか調べる
   - あるなら 2.へ
   - ないなら aに関するデータ依存関係はない。次の参照されている変数について 1.からやり直す。
2. 実行系列を一つ遡る
   - 遡れないならば aに対する探索は終了である
3. その実行時点で定義されているポインタ変数を探す
   - あるなら 4.へ
   - ないなら 2.へ
4. 定義されているポインタ変数が何を参照しているか調べる。ついている * の数だけ参照しているアドレスをたどる。たどる過程で aが
   - 現れたらデータ依存関係がある。
   - 現れないなら 2.へ

これと、第 4.6章のアルゴリズムを組み合わせることによりポインタを含むダイナミックスライシングができるようになる。

4.9 実装

今回は、実行系列に関するデータベースを作成する部分と、データ依存関係を追跡する部分だけ実装を行った。sint で実行できかつ、単一関数で、逐次実行のみからなるポインタを含まないプログラムに関するダイナミックスライスならば求めることができる。今回実装したダイナミックスライサの概念図を図 4.5に示す。

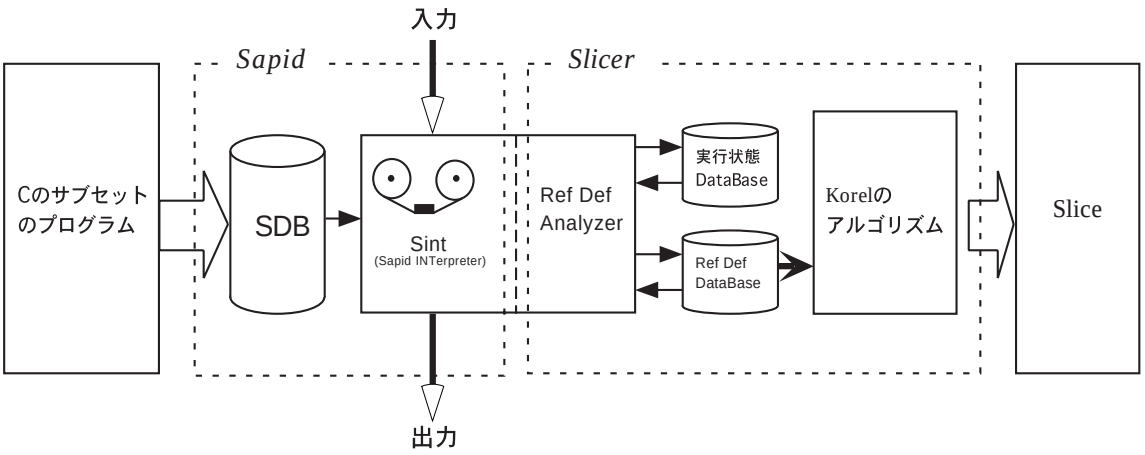

図 4.5: ダイナミックスライサの概念図

第 5 章

おわりに

5.1 実装

今回は、実行系列に関するデータベースを作成する部分と、データ依存関係を追跡する部分だけ実装を行った。sint で実行できかつ、単一関数で、逐次実行のみからなる、ポインタを含まないプログラムに関するダイナミックスライスならば求めることができる。

ソースコードは全部で 624 行。そのうちヘッダファイルが 87 行で、残り 537 行がコメントとソースコード。リスト操作などのスライシングの本質的ではないソースコードが 131 行、実行系列をえるために書いたソースコードが 108 行、データ依存関係をたどるために書いたソースコードが 156 行、などである。

機能ごとの独立性を考慮して設計されているので、制御依存関係や命令同一関係をたどるための機能を追加しても、今までのソースコードに大きな変更は加えなくても済むように考えてある。例えば、命令同一関係をたどるにはそのための関数を独立に実装し、データ依存関係をたどる関数の前か後に加えればいいはずである。

5.2 Sapid の評価

今回、実装に Sapid を用いた。Sapid を用いることにより実装の省力化が可能だったことは確認できた。今回改造した C 言語のインタプリタ sint は C 言語のプログラムで、その行数はコメント等すべてを入れて 104 行である。一方 Sapid を用いていない C 言語のインタプリタに CINT[Got99] があるが、これのソースコードは主な部分だけで 85041 行ある。機能に大きな違いがあるので一概にはいえないがこの差は大きい。

今回の実装に当たっては、ソースコードについて意識することはいっさいなかった。Sapid

が提供するデータベースSDBの上だけで設計と実装について考えることができた。したがってSDBを作る過程の字句解析や構文解析についていっさい考える必要はなかった。また、SDBのアクセスもARで提供されている関数で十分だった。

一方でC言語のインタプリタであるsintの完成度は高いとは言えなかった。実現されていない機能がかなり目についた。sintが動かないのは確認してる限りでは次のような機能である。

- ほとんどのライブラリ関数 (一部は動作している)
- ほとんどのシステムコール (一部は動作している)
- 構造体のメンバへのアクセス (演算子".")
- 構造体へのポインタからメンバへのアクセス (演算子"->")
- 配列を使うとBus Error

この中で、ライブラリ関数やシステムコールが動作しないのはsintが用いてる動的解析のためのライブラリSIP2が対応していないためである。SIPがもっと正しく動作すれば、見かけだけでも正しく動作すれば、Sapidによる動的解析はかなり有効なことが予想できる。ライブラリの部分だけでもうまく対処すればかなりの成果が期待できる。
例)

- システムコールの呼び出しは失敗しないと仮定する。
- ライブラリ関数の返り値と引数の間にあるデータ依存関係のデータベースを作る
- ライブラリ関数がポインタ渡しの場合は参照している値を変更するかどうかの情報を与える。

などが考えられる。

動的解析にSapidを応用する場合、今回の実装のように提供されたインタプリタに何らかの機能を付加して解析を実行するという手法が多く取られることが考えられる。ぜひ、インタプリタの機能向上を実現してほしい。

Sapidの大きな問題点として習得が困難なことが上げられる。その理由はいくつか考えられる。

- ドキュメントが理解しにくい

  - 個々のAPIに関するドキュメントはあるが、全体の中での位置づけや見通しが悪い。
  - ドキュメントの量が少なく、ここの機能の全体像がつかめない。そもそもドキュメントがないことすらある。

  Sapidには様々な機能が用意されている。そして、その機能ごとにドキュメントが用意されている。しかし、ドキュメントを読んでもその内容が理解できないことが多かった。特に、API群に関するドキュメントの不親切さは、開発に致命的な影響を与えた。Sapidは開発期間が長く、その間に蓄積されたドキュメントの整理が困難な事情は想像がつくが、そのためにSapidの利便性が著しく低下しているように思えてならない。

- ソースコードが読みにくい
  疑問に思ったことがドキュメントでは解決できない場合ソースコードを読みに行くことになるが、それが読みにくかった。どちらかは整備されることが望まれる。

  - コメントがほとんど書いていない。
  - オブジェクト指向を意識して、下層を隠蔽するような実装が心掛けられているが、ドキュメントやコメントがないせいで、逆に見通しの悪い実装になってしまっている。

- 規模が大きい
  Sapidは含まれる様々なAPIやツールにより規模が非常に大きくなっている。例えば、解凍した直後のSapidのパッケージには1411個のファイルが含まれており、合計で31MBのサイズがある。前出のCINTは976個のファイルからなり合計で13MBのサイズである。そのため全体像の把握が困難になってしまっている。似たような機能を持つものが複数存在したり、もう使われなくなっているものがあるなどしている。見通しを良くするために機能やツールの取捨選択や統合が望まれる。

5.3 結論

本研究では以下の三つの結論を得た。ダイナミックスライシングにおける：

- Sapidの有効性を部分的に確認した
- Sapidの問題点を指摘した

- 実装面から問題点を指摘した

今回Sapidを二つの側面から利用した。構文情報データベースとそこにアクセスする手段という側面とソースコードの実行環境という側面である。ダイナミックスライシングでは両方の側面が要求される。

最初の側面に対してSapidは非常に良く機能した。実行されたソースコードに対しては変数の定義参照情報データベースをいとも簡単に作り上げることができた。わずか56行である。

もう一方の実行環境という側面ではSapidは満足な成果を出せなかった。実行環境として選んだsint(Sapid INTerpreter)が不正確にしか動的情報を得られなかったのである。ライブラリ関数が十分に実装されていないこと、複雑なポインタが扱えないこと、構造体と配列が正しく扱えないことなどが理由である。

一方、ダイナミックスライサの実装という観点では、これまでも度々言われてきているキャストや高階ポインタの追跡が問題になった。定義参照関係が曖昧になってしまうのである。定義参照関係を追跡するために各実行時点の変数の状態をすべて保存しようとすると多くのメモリが必要になってしまうのも問題である。また、ライブラリ関数もデータ依存関係の追跡を断切ってしまうので問題である。

謝辞

本研究を進めるに当り、最後まで粘り強く御指導を頂いた権藤克彦助教授に深く感謝致します。数々の助言、叱咤激励を頂いた片山研究室の諸先輩方に厚く御礼申し上げます。本論文をまとめるに当たり助言や暖かい励ましを頂いた片山研究室及び権藤研究室の皆様に心から御礼申し上げます。

参考文献

[Agr] Hiralal Agrawal. Dynamic slicing in the presence of unconstrained pointers. Technical report.

[BJ88] B.Korel and J.Laski. Dynamic program slicing. *Information Processing Letters*, Vol. 29, No. 10, pp. 155–163, October 1988.

[BJ90] B.Korel and J.Laski. Dynamic slicing of computer programs. *Journal of Systems Software*, No. 13, pp. 187–195, 1990.

[Got99] Masaharu Goto. The cint c/c++ interpreter. http://hpsalo.cern.ch/root/Cint.html, oct 1999.

[HJ90] H.Agrawal and J.R.Horgan. Dynamic program slicing. *ACM SIGPLAN Notices*, Vol. 25, No. 6, pp. 246–256, June 1990.

[HS] Susan Horwitz and Marc Shapiro. Modular pointer analysis.

[SH97a] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *4th International Symposium on Static Analysis*, LNCS 1302. Springer-Verlag, September 1997.

[SH97b] Marc Shapiro and Susan Horwitz. Fast and accurate flow insensitive points-to analysis. *ACM symposium on Principles of Programming Language*, January 1997.

[Ste95] Bjarne Steensgaard. Points-to analysis in almost linear time. *ACM symposium on Principles of Programming Language*, pp. 32–41, 1995.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pp. 352–357, July 1984.

[WP99] Brian W.Kernighan and Rob Pike. *The Practice of Programming.* Addison-Wesley, 1999.

[YHR99] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. *ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[下村 95] 下村隆夫. プログラムスライシング技術と応用. 共立出版, July 1995.

[吉田 95] 吉田敦, 山本晋一郎, 阿草清滋. Case ツール開発のためのソフトウェア操作言語. 情報処理学会誌, Vol. 36, No. 10, pp. 2433–2441, October 1995.

[吉田 97] 吉田敦, 山本晋一郎, 阿草清滋. 意味を考慮した差分抽出ツール. 情報処理学会誌, Vol. 38, No. 6, June 1997.

[吉田 98] 吉田敦, 山本晋一郎, 阿草清滋. ソースプログラムに対する変更操作が可能な細粒度ソフトウェアリポジトリの提案. 日本ソフトウェア科学会 FOSE, pp. 189–198, 1998.

[橋本 94] 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. 電子情報通信学会技術研究報告 SS94, Vol. 10, pp. 73–80, May 1994.

[山本 95] 山本晋一郎, 阿草清滋. 細粒度リポジトリに基づいたツール・プラットフォームとその応用. 情報処理学会ソフトウェア工学研究会, Vol. 102, No. 7, pp. 37–42, 1995.

[大崎 96] 大崎博基, 山本晋一郎, 阿草清滋. プログラム理解のための依存関係表示ツール. 日本ソフトウェア科学会 FOSE, pp. 1163–1171, 1996.

[日本] 日本ラショナルソフトウェア株式会社. PURIFY Version 4.0.1 機能評価ガイド.

[福安] 福安直樹, 山本晋一郎. *Sapid: Sophisticated APIs for CASE tool Development.*

[福安 98a] 福安直樹, 山本晋一郎, 阿草清滋. 細粒度ソフトウェア・リポジトリに基づいたソースプログラムの安全な変更. コンピュータソフトウェア, Vol. 98, No. 4, pp. 78–81, July 1998.

[福安 98b] 福安直樹, 山本晋一郎, 阿草清滋. 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. 情報処理学会誌, Vol. 39, No. 6, June 1998.

[木野 96] 木野和佳, 山本晋一郎, 阿草清滋. プログラム動作理解のための抽象実行系. 日本ソフトウェア科学会 FOSE, pp. 98–101, 1996.

[有賀] 有賀寛朗, 山本晋一郎. Sapid I-model ver. 4.2 仕様書.