ソフトウェア工学 106－6
（1995．11．17）

依存関係が定義可能なテストベッド

内山 晃司　山本 晋一郎　阿草 清滋

名古屋大学 工学部 情報工学科

〒 464-01 名古屋市千種区不老町

数多く提案されているホワイトボックス法によるテストを容易に実現するためには、ソフトウェアの制御およびデータの依存関係のモデル化が必要である。本稿では、制御とデータの依存関係を取得するためのライブラリ (SDA) について述べる。SDA では、これらの依存関係を有向グラフを用いて表現し、グラフへのアクセス関数を提供する。SDA を用い依存関係を取得することにより、テストツールのプログラマは依存関係の解析の負担が軽減され、ツールのアルゴリズムの本質的な部分の記述に専念することができる。また SDA を用いて、既存のテスト基準に基づいたテストツールの構築を行なった。本稿では、全分岐網羅の基準に基づいたツールを例として示す。

A Dependency Definable Test Bed

Koji UCHIYAMA　Shinichirou YAMAMOTO　Kiyoshi AGUSA

Department of Information Engineering, Nagoya University

Furou-cho, Chikusa-ku, Nagoya-shi, Aichi 464-01

In this paper, we show Sapid Dependency Analyzer called SDA, which is a library for dependency analysis. To implement various testing method of software, it is necessary to model the control dependency and the data dependency of programs. SDA expresses the dependencies as directed graphs, and provides access routines to access them. SDA enables test tool programmers to concentrate the implementation of test algorithm. We have implemented testing tools based on existing testing criteria, and we show a tool based on C1 coverage as an example.

1 はじめに

近年の計算機の発達にともない、大規模なソフトウェア開発の要求が増している。これにともない、ソフトウェアの開発のライフサイクルの工程 (phase) の一つであるテスト工程においても、効率の向上が切望されている。ソフトウェアのテストの効率化は、開発期間を短縮するための重要な要素である。

ソフトウェアのテスト技法のうち、ホワイトボックス法によるモジュールテストでは、命令、分岐、実行経路に基づくテスト基準が提案されている。しかし、これらのテスト基準はどれも一長一短で、決定的なものが存在しない。また、これらを拡張したものとして、データの流れに基づいたテスト基準が提案されている。

これらの基準に基づいたテストツールを実現するときには、プログラムの制御の依存関係、データの依存関係を求める部分の作成が必要不可欠である。

また、C 言語のようにポインタ変数を含む言語では、別名解析 (alias analysis) を行ない、実行時にポインタ変数が指す可能性のある変数を求める必要がある。別名解析では、プログラムの制御の依存関係と、ポインタ変数を変数とみなしたデータの依存関係の取得が必要である。

そこで、プログラムの制御およびデータの依存関係をモデル化することは、テストツールの実現を簡潔にし、プログラマの負担を軽減する。

本稿では、ホワイトボックス法のモジュールテストのツールを簡潔に記述するための環境として、プログラムの依存関係として制御とデータの依存関係を扱うライブラリ (SDA: Sapid Dependency Analyzer) について述べる。

SDA は C 言語のライブラリとして提供され、我々が開発している細粒度リポジトリ [1](Sapid) 上で稼働する。

SDA はソフトウェアのテストだけではなく、コンパイラの最適化手続き、部分評価、プログラムスライシングなどの CASE ツールでの、プログラムの制御や、変数の依存関係を扱う部分でも用いることができる。制御の依存解析、データの依存解析を行なう部分を細粒度リポジトリ側で行ない、これらのアクセス関数を提供することは、プログラマの負担を軽減する (図 1)。

2 Sapid の概要

Sapid(Sophisticated Application Programming Interface for software Development) は、細粒度のソフトウェアリポジトリに基づいたツールプラットフォームである。

Sapid はソフトウェアデータベース (SDB: Software Database)[2]、アクセスルーチン (AR: Access Routines)、ソフトウェア操作言語 (SML: Software Manipulating Language)[5] からなる。

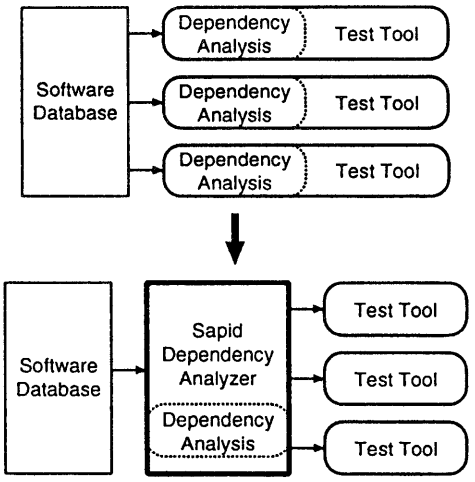

図 1: SDA

SDB は、対象となるソフトウェアをソフトウェアモデルに基づき解析し、得られた実体、関連情報を格納するデータベースである。

ソフトウェアの実体を project, program, file, function, declaration, block, label, expression, funccall, const, member とし、これらの構成関係、定義参照関係を関連としたのが、我々のソフトウェアモデルである。

前処理後の C 言語のプログラム (.i プログラム) に対してのモデルは Sapid Imodel(以下 Imodel) と呼ばれる。

AR は、SDB への基本的なアクセスルーチンで、データベースオブジェクトの属性値の取得、オブジェクトの関連の取得が可能である。AR を用いると、プログラマは基本的な情報の取得や、簡単なソースプログラムの変更を行なうことができる。

SML はソフトウェアのより高度な操作を行なうための言語である。SML は、オブジェクト指向言語 Sather[4] へのトランスレータとして実現されている。

また、Sapid を用いた応用プログラムとして、SDB の識別子の情報をもとにして、Emacs エディタ上での置換を行なうソフトウェア操作エディタ、仕様書とソースプログラムの整合性を保つ作業、あるいは整合性をチェックする作業を自動化する仕様管理支援システム、プログラム中のある文に関係するすべての文を抽出する program slicing ツールの試作 [3] が実現されている。

3 SDA

ソフトウェアの効率的なテストを行なうためには、制御の依存関係、データの依存関係など、Imodel より抽

象度が高い関係を扱う必要がある。

SDA では、Imodel より高レベルなソフトウェアモデルを用いる。SDA では、ソフトウェアの依存関係として、制御およびデータの依存関係を扱う。

SDA の構成を図 2に示す。

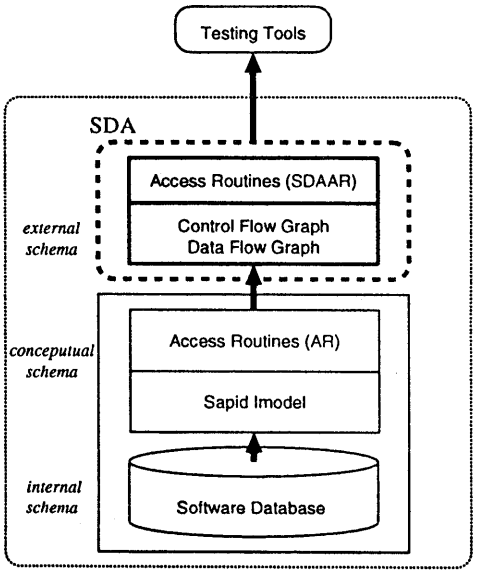

図 2: SDA の構成

データベースとしての細粒度リポジトリに対し、Imodel が概念スキーマ (conceptual schema) であり、これを実現したのが AR である。

SDA では外部スキーマ (external schema) として、プログラムの制御およびデータの依存関係に基づいたモデルを用いる。

SDA で扱う依存関係は、ソフトウェアにおける制御の依存関係と、データの依存関係である。これらの依存関係は有向グラフで表現される。制御の依存関係は制御流れグラフ (CFG: control flow graph) として、データの依存関係はデータ流れグラフ (DFG: data flow graph) として表現される。

CFG と DFG 間では、これらのグラフどうしの節点間の関係が取得可能である。

SDA では制御の依存関係は CFG で表現する。CFG ではプログラムまたは関数を、これらを構成するステートメントを節点 (node) としてあらわし、また 2 節点間の制御の依存関係を、2 つの節点を結ぶ辺 (edge) であらわす。

SDA ではデータの依存関係は DFG で表現する。しかし、データの依存関係の定義はその目的によってさまざまに変わり得る。そこで、SDA ではデータの依存関係をユーザが定義可能とする。様々なデータの依存関係の定義に基づき、グラフを生成する関数をそれぞれ作り、テストツールのプログラマは、グラフを生成する関数を選択できるようにする。すなわち、DFG における辺の意味は、グラフの生成関数によって与えられる。また、特殊な依存関係については必要とされる依存関係を手続きとして外部から与えると、与えられた関係の定義に基づき DFG を生成することができる。

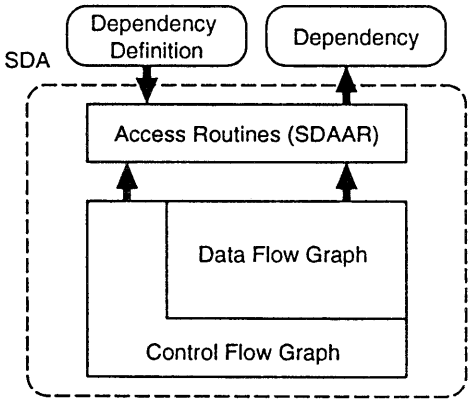

図 3: SDA の内部モデル

C 言語のようにポインタを含む言語では、データの依存関係の解析に別名 (alias) の解析が必要となる。そこで、テストツールのプログラマは、ポインタを含む言語に関しては、必要に応じて別名解析部 (alias analyzer) を用いながら、テストツールを作成する。別名解析部は現在、試作の段階である。

また、プログラムのテストツールのアルゴリズムの特徴として、CFG や DFG の節点または辺の集合や、集合に対する演算が頻繁に用いられる点がある。このため、SDA は節点や辺の集合演算関数を提供する。

4 制御の依存関係

プログラムの制御の依存関係を取得するためには、プログラムを CFG によってモデル化し、CFG のアクセス関数を用いて実現する。この節では SDA の CFG の構造について述べる。

4.1 CFG の構造

CFG は節点 (node) の集合 N_c と辺 (edge) の集合 E_c の 2 項組 (N_c, E_c) で表される。

4.1.1 CFG の節点

CFG の節点はプログラムのステートメント、条件分岐、ラベル、関数の入口、関数の出口のいずれかに対応する。節点は以下の属性を持つ。

- 節点の種類 (以下のうちのいずれか)
  - 入口 (entry node)
  - 出口 (exit node)
  - ステートメント (statement node)
  - 分岐 (if node)
  - ラベル (label node)
- ステートメントに対応する Imodel の expression クラスのオブジェクト
- 分岐の節点における、分岐条件の式 (expression) に対応する Imodel のオブジェクト

節点は、種類によって分類され、それぞれ開始節 (entry node)、終了節 (exit node)、ステートメント節 (statement node)、条件節 (if node)、ラベル節 (label node) と呼ぶ。

4.1.2 CFG の辺

CFG の辺は、節点間の制御の依存関係を表わす。
CFG の辺は以下の属性を持つ。

- 開始点の節点
- 終了点の節点
- 辺の種類 (「無条件」、「真」、「偽」)

辺の種類の属性において、属性値「無条件」は制御が無条件で次の節点に移ることを意味し、属性値「真」、「偽」を持つ辺は開始点が「分岐」の属性を持つ辺で、条件がそれぞれ真、偽のときに制御が移ることを意味する。

プログラムの制御構造を SDA のモデルで表現するとき、繰り返し構造は、すべて分岐として表現される。これは、CFG の解析を容易にするためである。

5 データの依存関係

プログラムのデータの依存関係の取得は、プログラムを DFG を用いてモデル化し、DFG へのアクセス関数で実現する。

データの依存関係はアプリケーションプログラムの用途に応じさまざまである。このため、SDA ではデータの依存関係の定義をテストツールのプログラマが手続き的に与えることにより、与えられた定義に基づく DFG を生成することが可能である。

テストツールではデータの依存関係として、

- 定義 – 使用の関係
- 代入の関係

が頻繁に用いられる。データの流れに基づくテスト方法では、変数の右辺値 (r-value)、左辺値 (l-value) と定義 – 使用の関係の取得が必要である。またポインタを含む言語では、定義 – 使用の関係を別名を用いて解析する必要があり、この解析には代入の関係の取得が必要である。

このため、上の関係に基づいた DFG の生成部については SDA が提供する。

テストツールのプログラマは、上記の一般的な依存関係については、SDA で提供される関数を用い、特殊なものについてはプログラマが定義を与える。

5.1 DFG の構造

DFG は節点の集合 N_d と辺の集合 E_d の 2 項組 (N_d, E_d) で表現される。

DFG の節点は変数の出現 (occurence) に対応する。節点は、対応する変数の Imodel の expression クラスのオブジェクト、変数に対応する CFG の節点の属性を持つ。

辺はデータの依存関係に対応する。辺の属性は、依存元の節点、依存先の節点である。

5.2 CFG, DFG の例

図 4に示したプログラムに対する、CFG、DFG の例を図 5に示す。図 5の左が CFG で、右が DFG である。DFG の節点に対応する変数の出現は、左にある CFG の節点での変数の出現に対応する。この例での DFG の依存関係は、定義 – 使用の関係である。

5 行目の `a = 1;` では、変数 a が定義されている。この定義が使用されるのは、11, 13 行目の `printf(...);` 文である。スコープの異なる a が 8 行目で定義され、9 行目で使用される。また、14, 17 行目で定義された変数 a が 19 行目で使用される。この例での DFG は、この定義 - 使用関係が辺に対応している。

定義 – 使用の関係に基づいた DFG を用いると、変数の右辺値、左辺値を容易に取得することができる。

6 SDAAR

SDAAR は CFG、DFG にアクセスするための API 関数である。SDAAR は大きく分けて初期化関数、アクセス関数に分類される。

6.1 初期化関数

CFG や DFG に対するアクセス関数を使うためには、初期化が必要である。初期化を行なうことにより、SDA

```
1 void function(void)
2 {
3     int a;
4
5     a = 1;
6     {
7         int a;
8         a = 22;
9         printf("1: a = %d \n", a);
10    }
11    printf("2: a = %d \n", a);
12    if (condition()) {
13        printf("3: a = %d \n", a);
14        a = 333;
15    }
16    else {
17        a = 4444;
18    }
19    printf("4: a = %d \n", a);
20 }
```

図 4: 被解析プログラム

は被解析プログラムの CFG を、また必要に応じ DFG を生成する。DFG の生成については、用途にあった依存関係の指定も行なう。

現在は、ソースプログラムの同じ変数名の出現に対し、それぞれの出現を DFG の節点とし、変数が左辺に現れたときには「定義 (def)」右辺や関数の引数として現れたときには「使用 (use)」に分類し、同じ変数名での定義 - 使用関係を依存関係として扱っている。

6.2 CFG に対するアクセス関数

CFG に対するアクセス関数は大きく分けて節点、辺を返す関数、判定結果を返す関数に分類される。

6.2.1 節点を返す関数

節点を返す関数としては以下のものがある

- CFG のすべての節点を返す関数
- CFG の入口の節点 (entry node) を返す関数
- CFG の出口の節点 (exit node) を返す関数
- 指定された節点の先行節 (predecessor) を返す関数
- 指定された節点の後続節 (successor) を返す関数

6.2.2 判定結果を返す関数

以下に、判定結果を返す関数について述べる。

- 節点 from から、節点 to までの到達可能性を返す関数。

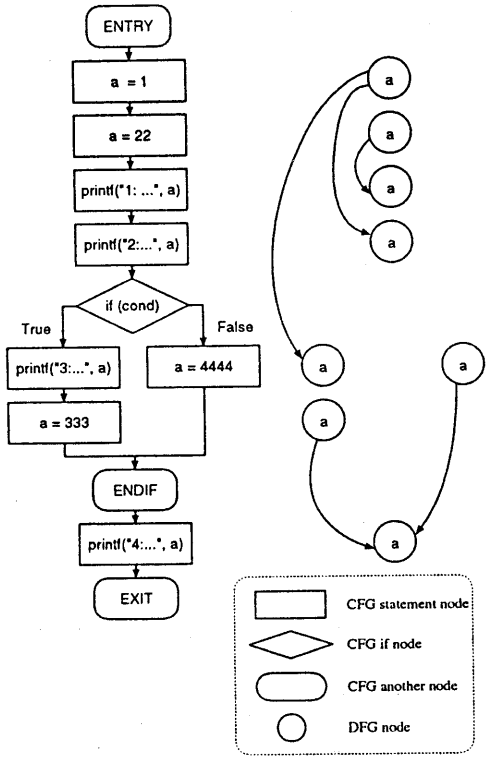

図 5: CFG, DFG の例

6.3 DFG に対するアクセス関数

DFG に対するアクセス関数は、大別して以下のものがある。

- 指定された節点の依存先の節点を返す関数
- 指定された節点の依存元の節点を返す関数
- 指定された 2 節点間の依存関係の有無を返す関数

6.4 その他の関数

6.4.1 集合を扱う関数

SDA では CFG, DFG の節点および辺の集合を扱うことができる。以下に集合を扱う関数を示す。

- 集合の生成、削除
- 要素の追加、削除
- 集合どうしの演算 (積, 和集合, 差)
- 集合に含まれる要素の取得
- 要素が集合に含まれるかの判定

7 実現および使用例

この節では、4, 5節で述べたCFG、DFGの実現と、SDAの使用例について述べる。

7.1 SDAの実現

SDAはSapidのパッケージで、C言語のライブラリとして提供され、現在、C言語のソースプログラムを解析の対象をしている。SDAのソースコードは約2900行である。

CFGの生成は、再帰降下法で行なう。このため、グラフの生成のコストは、ソースプログラムに対するImodelのオブジェクト数 n に対し、$O(n)$ である。

7.2 制御文の扱い

C言語の制御文のうち for, do-while のCFGの例を図6に示す。

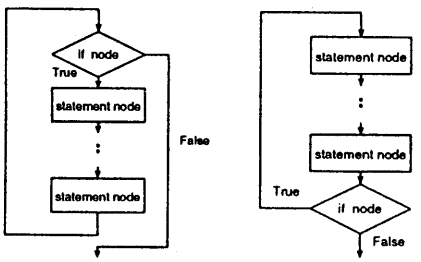

図 6: 制御文の実現 (左: for, 右: do-while)

この他の制御文 if, while, switch についても、同様に分岐の節点として表現される。

8 カスタマイズ可能なテスト環境

ソフトウェアのテスト技法のうち、ホワイトボックス法によるモジュールテストでは、命令、分岐、実行経路に基づくテスト基準が提案されている。これらの基準はそれぞれ、全命令網羅 (C_0coverage)、全分岐網羅 (C_1 coverage)、全経路網羅 (以下 C_P) と呼ばれる。これらの基準の強さは $C_0 \leq C_1 \leq C_P$ の関係がある。

基準 C_1 は一般的に用いられている基準の一つであるが、実行経路に依存して引き起こされるエラーは発見できないので、基準としてはものたりない。

基準 C_P は、上で挙げたエラーを発見できる一方、基準が厳し過ぎるために、プログラムが大きくなると、すぐに基準を満たすべき経路の数が膨大になってしまう。

そこで、実行時に通り得るすべての経路の中から、何らかの手法を用いて、エラーが含まれる可能性の高い経路をテストすることが必要となる。

この手法の一つとして、ソフトウェアのデータの流れに基づいたテストの研究がなされている [6]。文献 [6] では、変数が定義されてから使用されるまでの経路中にエラーが含まれる可能性が高いという仮定に基づいたテスト基準が提案されている。

このようにソフトウェアの制御、データの依存関係に基づいたテスト基準が多く提案されている。

SDAを用いると、上で挙げたテスト基準に基づいたテストツールが簡潔に記述可能となる。

例として、基準 C_1 に基づいたテスト環境のプログラムの例を図7に示す。このプログラムは、テスト対象の関数中のすべての分岐文の分岐先の先頭に、実行時の通過情報を出力するコードを付加するプログラムである。

プログラム中、名前が sda で始まる関数はSDAのAPI関数で、spd で始まる関数はARのAPI関数である。sdaGetAllCEdges() は、すべてのCFGの辺を取得する関数である。add_insertion_list() は、分岐先の先頭に対し、実行時の通過情報を挿入リストに登録する関数で、insert() は、このリストに基づき挿入を行なう関数である。file_out() は、file_buf の内容を標準出力に出力する関数である。

このプログラムの総行数は約200行である。テスト基準に対する網羅度はこの出力を集計して算出される。

```
 1 void add_log(int funcId, char *filename)
 2 {
 3   SdaCursor   *csr;
 4   SdaCEdge    *ce;
 5   SdaCFG      *cfg;
 6   int         offset, length, stmtId;
 7   char        s[MAX_OBJ_LENGTH],
 8               file_buf[MAX_FILE_SIZE];
 9
10   cfg = sdaMakeCFG(funcId);
11   filein(file_buf, filename);
12   csr = sdaGetAllCEdgesInit(cfg);
13   while ((ce = sdaGetAllCEdges(csr)) != NULL) {
14     if ((ce->edgeType == CEDGE_TRUE) ||
15         (ce->edgeType == CEDGE_FALSE)) {
16       stmtId = ce->to->stmtId;
17       offset = GetCOffsetOfObj(stmtId);
18       sprintf(s, TEST_COMMAND "\n%s",
19               ce->to, spdGetObjText(stmtId));
20       add_insertion_list(offset, s);
21     }
22   }
23   sdaFreeCursor(csr);
24   insert(file_buf);
25   fileout(file_buf);
26 }
```

図 7: 変換プログラム

図4のプログラムを入力したときの出力結果を図8に示す。ただし見やすくするため整形してある。

このように、上で挙げた他の基準に関してもテストツールを簡潔に記述することができる。また現在、データの流れに基づいたテスト基準に、別名解析を利用し経路を減らす方法を考察中であり、SDAはこれらの基準に関しても、簡潔に記述できる。

```
 1 void function(void)
 2 {
 3     int a;
 4
 5     a = 1;
 6     {
 7         int a;
 8         a = 22;
 9         printf("1: a = %d \n", a);
10     }
11     printf("2: a = %d \n", a);
12     if (condition()) {
13         sda_log("Node 100103a0 ");
14         printf("3: a = %d \n", a);
15         a = 333;
16     }
17     else {
18         sda_log("Node 10010260 ");
19         a = 4444;
20     }
21     printf("4: a = %d \n", a);
22 }
```

図 8: 変換されたプログラム

9 おわりに

SDA を用いることによって、従来の方法とくらべて、より簡単にアルゴリズムを記述することができた。これを用いることにより、テストドライバが簡潔に記述できる見通しが立った。

現在分かっている問題点をあげる。

9.1 アクセス関数の必要十分性の検討

CFG および DFG へのアクセス関数がどれだけ必要とされて、またどのくらい与えれば十分であるかの判定は、経験に基づいておこなったが、これの妥当性の証明はなされていない。

9.2 他のテスト方法への対応

本稿では、テスト方法をホワイトボックス法によるモジュールテストに限定したが、他のテスト方法にも対応できるよう拡張する必要がある。

参考文献

[1] 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. 95-SS-102, pp. 37–42, Jun 1995.

[2] 有賀寛朗, 山本晋一郎, 阿草清滋. ソフトウェア構造解析情報に基づくツールプラットホームシステム. 電子情報通信学会 ソフトウェアサイエンス研究会, Vol. SS94, No. 15, pp. 25–32, July 1994.

[3] 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, No. 10, pp. 73–80, May 1994.

[4] Stephen M. Omohundro. The Sather Language Technical report, Internal Computer Science Institute, 1991.

[5] 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 論文誌 Vol. 36, No. 10, 1995(掲載予定)

[6] Sandra Rapps, Elaine J.Weyuker, "Selecting Software Test Data Using Data Flow Information" *IEEE Trans. Software Eng.*, vol. SE-11, NO.4 pp.900-910, April 1985.

[7] T.J. Marlowe, W.G. Landi, B.G. Ryder, J.D. Choi, M.G.Burke, and P. Carini. "Pointer-induced aliasing: A clarification." *SIGPLAN Notices, 28(9)* pp.67-70, Sept. 1993.

[8] William Landi, Barbara G.Ryder "A safe approximate algorithm for interprocedural pointer aliasing" *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp.235-248, 1992. Published as SIGPLAN Notices, 27(7)

[9] J. D. Choi, M.G. Burke, and P. Carini. "Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pp.232-245. Association for Computing Machinety–SIGPLAN, Jan. 1993.