BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC DÂN LẬP HẢI PHÒNG

-------o0o-------

ISO 9001:2000

ISO 9001 : 2008

ĐỒ ÁN TỐT NGHIỆP

Ngành Công nghệ Thông tin

BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC DÂN LẬP HẢI PHÒNG

-------o0o-------

NÂNG CAO CHẤT LƯỢNG PHẦN MỀM BẰNG CÁC KỸ THUẬT PROGRAM SLICING

ĐỒ ÁN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY

Ngành: Công nghệ thông tin

NHIỆM VỤ THIẾT KẾ TỐT NGHIỆP

Sinh viên: Nguyễn Sỹ Linh Mã số: 1351010032

Lớp: CT1301 Ngành: Công nghệ thông tin

Tên đề tài:

NÂNG CAO CHẤT LƯỢNG PHẦN MỀM BẰNG CÁC KỸ THUẬT PROGRAM SLICING

NHIỆM VỤ ĐỀ TÀI

1. Nội dung và các yêu cầu cần giải quyết trong nhiệm vụ đề tài tốt nghiệp

a. Nội dung:

- Nắm được các khái niệm cơ bản về program slicing
- Nắm được các phương pháp trong program slicing
- Thử nghiệm trên một số chương trình đơn giản

b. Các yêu cầu cần giải quyết

.....
.....

.....
.....

.....
.....

.....
.....

.....
.....

2. Các số liệu cần thiết để thiết kế, tính toán

.....
.....

.....
.....

3. Địa điểm thực tập

CÁN BỘ HƯỚNG DẪN ĐỀ TÀI TỐT NGHIỆP

Người hướng dẫn thứ nhất:

Họ và tên: Nguyễn Trịnh Đông

Học hàm, học vị: Thạc sĩ

Cơ quan công tác: Khoa Công nghệ Thông tin – Trường Đại Học Dân Lập Hải Phòng

Nội dung hướng dẫn: …….....

.....
.....
.....
.....
.....

Người hướng dẫn thứ 2:

Họ và tên: …….....

Học hàm, học vị:…….....

Cơ quan công tác: ……..

Nội dung hướng dẫn: …….....

.....
.....
.....
.....
.....

Đề tài tốt nghiệp được giao ngày … tháng … năm 20...

Yêu cầu phải hoàn thành trước ngày … tháng … năm20...

Đã nhận nhiệm vụ: Đ.T.T.N	Đã nhận nhiệm vụ: Đ.T.T.N
Sinh viên	Cán bộ hướng dẫn Đ.T.T.N

Hải Phòng, ngày …….. tháng …….. năm 20……

HIỆU TRƯỞNG

GS.TS.NGƯT Trần Hữu Nghị

PHẦN NHẬN XÉT TÓM TẮT CỦA CÁN BỘ HƯỚNG DẪN

1. Tinh thần thái độ của sinh viên trong quá trình làm đề tài tốt nghiệp:

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

2. Đánh giá chất lượng của đề tài tốt nghiệp (so với nội dung yêu cầu đã đề ra trong nhiệm vụ đề tài tốt nghiệp)

.....

.....

.....

.....

.....

.....

.....
.....

.....
.....

.....
.....

.....
.....

.....
.....

3. Cho điểm của cán hộ hướng dẫn:

(Điểm ghi bằng số và chữ)

.....
.....

.....
.....

.....
.....

.....
.....

Ngày ... tháng ….. năm 20…

Cán bộ hướng dẫn chính

(Ký, ghi rõ họ tên)

PHẦN NHẬN XÉT ĐÁNH GIÁ CỦA CÁN BỘ CHẤM PHẢN BIỆN
ĐỀ TÀI TỐT NGHIỆP

1. **Đánh giá chất lượng đề tài tốt nghiệp (về các mặt như cơ sở lý luận, thuyết minh chương trình, giá trị thực tế, ...)**

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

2. Cho điểm của cán bộ phản biện

(*Điểm ghhi bằng số và chữ*)

.....

.....

.....

.....

.....

.....

.....

.....

Ngày ... tháng ….. năm 20…

Cán bộ chấm phản biện

(Ký, ghi rõ họ tên)

LỜI CẢM ƠN

Trước hết em xin bày tỏ tình cảm và lòng biết ơn đối với thầy Nguyễn Trịnh Đông – Khoa Công nghệ Thông tin – Trường Đại học Dân Lập Hải Phòng, người đã dành cho em rất nhiều thời gian quý báu, trực tiếp hướng dẫn tận tình giúp đỡ, chỉ bảo em trong suốt quá trình làm đồ án tốt nghiệp.

Em xin chân thành cảm ơn tất cả các thầy cô giáo trong khoa Công nghệ Thông tin - Trường ĐHDL Hải Phòng, chân thành cảm ơn các thầy giáo, cô giáo tham gia giảng dạy và truyền đạt những kiến thức quý báu trong suốt thời gian em học tập tại trường, đã đọc và phản biện đồ án của em giúp em hiểu rõ hơn các vấn đề mình nghiên cứu, để em có thể hoàn thành đồ án này.

Em xin cảm ơn GS.TS.NGƯT Trần Hữu Nghị Hiệu trưởng Trường Đại học Dân lập Hải Phòng, Ban giám hiệu nhà trường, Bộ môn tin học, các Phòng ban nhà trường đã tạo điều kiện tốt nhất trong suốt thời gian học tập và làm tốt nghiệp.

Tuy có nhiều cố gắng trong quá trình học tập, trong thời gian thực tập cũng như trong quá trình làm đồ án nhưng không thể tránh khỏi những thiếu sót, em rất mong được sự góp ý quý báu của tất cả các thầy giáo, cô giáo cũng như tất cả các bạn để kết quả của em được hoàn thiện hơn.

Em xin chân thành cảm ơn!

Hải Phòng, ngày tháng năm 2013

Sinh viên

Nguyễn Sỹ Linh

MỤC LỤC

DANH MỤC HÌNH ẢNH

DANH MỤC CÁC TỪ VIẾT TẮT

Tên viết tắt	Diễn giải
PDG (Program Dependence Graph)	Đồ thị phụ thuộc chương trình.
CFG (Control Flow Graph)	Đồ thị luồng điều khiển .
REF (is the set of variables whose values are used)	Là tập các biến có giá trị được sử dụng.
DEF (is the set of variables whose values are changed)	Là tập các biến có giá trị được thay đổi.
INFL (range of influence)	Tầm ảnh hưởng.
MOD (variables that may be modified)	Biến có thể sửa đổi.
USE (variables that may be used)	Biến có thể sử dụng.
SDG (System Dependence Graph)	Đồ thị phụ thuộc hệ thống.
DU (Definition- Use)	Quan hệ Định nghĩa – Sử dụng.
TC (Test- Control)	Quan hệ Kiểm thử- Điều khiển.
IR (Identity- Relation)	Quan hệ Định danh.
DDG (Dynamic Dependence Graph)	Đồ thị phụ thuộc động.
RDDG (Reduced Dynamic Dependence Graph)	Đồ thị phụ thuộc *dynamic* rút gọn.
DDSG (Dynamic Denpendence Synthesis Graph)	Đồ thị tổng hợp phụ thuộc động.

GIỚI THIỆU

Trong sản xuất phần mềm, rất nhiều hoạt động được thực hiện như khảo sát, phân tích, thiết kế,… Trong đó kiểm thử, bảo trì phần mềm là những công việc có tầm quan trọng nhằm đảm bảo phần mềm hoạt động chính xác, hiểu quả. Tìm hiểu về *Program Slicing* là một trong những phương pháp nâng cao chất lượng phần mềm.

Trong quá trình tìm hiểu tài liệu, em đã nghiên cứu, tìm hiểu và trình bày trong đồ án các phương pháp áp dụng trong *Progam Slicing* nhằm làm rõ một số kỹ thuật được áp dụng trong đó.

Các tác giả như *Weiser, Ottenstein, B. Korel*, J. Laski, *Horwitz,Reps,Binkley* …. Là những người tiên phong nghiên cứu về lĩnh vực này. Các tác giả đã đưa ra các khái niệm như *Static Slicing*, *Dynamic Slicing*, đồ thị phụ thuộc chương trình, đồ thị phụ thuộc luồng điều khiển,đồ thị phụ thuộc hệ thống… Từ đó hình thành một lĩnh vực nghiên cứu mới trong việc đảm bảo chất lượng phần mềm.

Với kiến thức như vậy, trong đồ án này em cũng chỉ bước đầu tìm hiểu các kỹ thuật đã được đề xuất để từ đó hình dung rõ hơn về một khía cạnh trong lĩnh vực sản xuất phần mềm. Do vậy, em chọn đề tài: "Nâng cao chất lượng phần mềm bằng kỹ thuật *Program Slicing*". Đồ án được trình bày như sau:

Giới thiệu: *GIỚI THIỆU VỀ BÀI TOÁN PROGRAM SLICING*

Chương 1: CÁC KHÁI NIỆM CƠ BẢN TRONG PROGRAM SLICING

Trình bày các khái niệm cơ bản được ứng dụng trong kỹ thuật Program Slicing.

Chương 2: CÁC KỸ THUẬT DÙNG TRONG PHƯƠNG PHÁP STATIC SLICING

Trong chương này trình bày về các kỹ thuật trong Static Slicing như: Phương pháp đơn thủ tục và đa thủ tục.

Chương 3: CÁC KỸ THUẬT DÙNG TRONG PHƯƠNG PHÁP DYNAMIC SLICING

Trong chương này trình bày về các kỹ thuật trong Dynamic Slicing như: Phương pháp đơn thủ tục và đa thủ tục.

Chương 4: *THỰC NGHIỆM TRÊN CÁC CHƯƠNG TRÌNH SLICER*

Chương này trình bày các công cụ trợ giúp trong việc nghiên cứu và tìm hiểu các kỹ thuật Program Slicing.

Kết luận: *Trình bày các kết quả tìm hiểu trong quá trình thực hiện đồ án.*

Cuối cùng là phần **Tài liệu tham khảo**.

Chương 1: CÁC KHÁI NIỆM CƠ BẢN TRONG PROGRAM SLICING

Theo Weiser [Wei1] *Program slicing* là một phương pháp được thực hiện giống như các lập trình viên có kinh nghiệm gỡ rối chương trình. Khi thực hiện các kỹ thuật trong gỡ rối chương trình, các lập trình viên thường lựa chọn các điểm cần quan tâm gọi là *Fix point.* Để từ đó dừng chương trình hoặc cô lập một số lệnh để gỡ rối. Công việc gỡ rối này đòi hỏi rất nhiều công sức cũng như thời gian. Nếu con người thực hiện thì có thể lại mắc những lỗi khác. Do vậy, có nhiều quan điểm ủng hộ việc tích hợp *program slicer* và môi trường gỡ rối.

Từ chương trình gốc, kỹ thuật *program slicing* sẽ tính toán để lựa chọn một tập các câu lệnh liên quan đến một biến hoặc một nhóm các biến nào đó để kiểm soát. Bắt đầu từ một tập hợp các hành vi của chương trình, cắt giảm chương trình mẫu tối thiểu mà vẫn tạo ra hành vi đó. Chương trình rút gọn, được gọi là một "*slice*", là một chương trình độc lập đảm bảo tính toàn vẹn của chương trình ban đầu.

Một *slice* của chương trình *P,* với tiêu chuẩn *slicing(n, v)* trong đó *n* là vị trí của câu lệnh trong chương trình (*n=1,..,tổng số câu lệnh*) và *v* là biến trong câu lệnh *n,* chỉ bao gồm các câu lệnh cần thiết của *P* để nắm bắt được hành vi của *v* tại câu lệnh *n*. Công việc tính toán được gọi là *program slicing*. Dưới đây là một số khái niệm hay dùng trong kĩ thuật *program slicing.*

1.1 Các định nghĩa

Định nghĩa 1: Đồ thị có hướng G

Đồ thị có hướng G là một cặp có thứ tự G:=(V, A), trong đó:

- *V, tập các đỉnh hoặc nút,*
- *A, tập các cặp có thứ tự chứa các đỉnh, được gọi là các cạnh có hướng hoặc cung. Một cạnh e = (x, y) được coi là có hướng từ x tới y; x được gọi là điểm đầu/gốc và y được gọi là điểm cuối/ngọn của cạnh.*

Định nghĩa 2: Đồ thị luồng điều khiển

Một đồ thị luồng điều khiển của chương trình P là một đồ thị trong đó mỗi nút tương ứng với một câu lệnh trong P và mỗi cạnh thể hiện cho

một luồng điều khiển trong P. Đồ thị luồng điều khiển(CFG) chứa hai nút đặc biệt là Start va Stop lần lượt thể hiện cho nút bắt đầu và kết thúc chương trình.

Định nghĩa 3: Lát cắt chương trình thực thi

Với câu lệnh thứ n và biến v, Một slice S của chương trình P đối với tiêu chuẩn slicing(n, v) là một chương trình thực thi bất kì với các đặc tính sau đây:

- *S có thể thu được bằng cách không xoá hoặc xoá nhiều câu lệnh từ P.*
- *Khi P dừng với một đầu vào cho trước thì S cũng phải dừng với đầu vào đó, P và S cùng tính toán ra một giá trị của các biến trong V khi câu lệnh tương ứng với nút n được thực hiện .*

1.2 Static slicing

Theo *Weiser, static slicing* được tính toán bằng cách xác định liên tiếp các tập hợp các biến liên quan của các câu lệnh theo sự phụ thuộc dữ liệu và phụ thuộc luồng điều khiển. Trong phương pháp này, chỉ những thông tin tĩnh được sử dụng tức là chỉ xem xét mã nguồn mà không quan tâm đến quá trình thực thi chương trình nên *slices slicing* của *Weiser* được gọi là *static slicing*.

Một phương pháp khác của *Ottenstein* sử dụng *đồ thị phụ thuộc chương trình PDG (Program)* để thực hiện chương trình. *PDG* là một đồ thị có hướng với các đỉnh tương ứng với các câu lệnh và tính chất điều khiển , các cạnh tương ứng là phụ thuộc dữ liệu và phụ thuộc điều khiển giữa các câu lệnh. Tiêu chuẩn *slicing* được xác định là một đỉnh trong *PDG* mà *slices clicing* bao gồm tất cả các đỉnh trong *PDG* mà từ đỉnh của tiêu chuẩn *slicing* có thể đi tới.

Một phương pháp khác được đưa ra bởi *Bergeretti* và *Carre* sử dụng quan hệ luồng thông tin từ một chương trình hướng cú pháp. Các phương pháp này tính toán tập các câu lệnh và xác định điều khiển bằng cách duyệt ngược chương trình bắt đầu từ tiêu chuẩn *slicing* nên *slices slicing* sinh ra gọi là *static slicing* ngược. *Bergeretti* và *Carre* sau đó giới thiệu *static slicing* tiến gồm tất cả các câu lệnh và tính chất điều khiển phụ thuộc vào tiêu chuẩn *slices*. Một câu lệnh bị phụ thuộc vào tiêu chuẩn *slices* nếu các giá trị được tính toán tại câu lệnh đó phụ thuộc vào giá trị được tính toán tại tiêu chuẩn *slices*, hoặc giá trị được tính toán tại tiêu chuẩn *slices* có tính chất quyết định sự thi hành của câu lệnh đó.

```
(1)    read(n);
(2)    i := 1;
(3)    sum := 0;
(4)    product := 1;
(5)    while i <= n do
       begin
(6)      sum := sum + i;
(7)      product := product * i;
(8)      i := i + 1
       end;
(9)    write(sum);
(10)   write(product)
```

(a)

```
read(n);
i := 1;

product := 1;
while i <= n do
begin

  product := product * i;
  i := i + 1
end;

write(product)
```

(b)

Hình 1: (a) Chương trình mẫu, (b) *slice slicing* của chương trình với tiêu chuẩn (10, product)

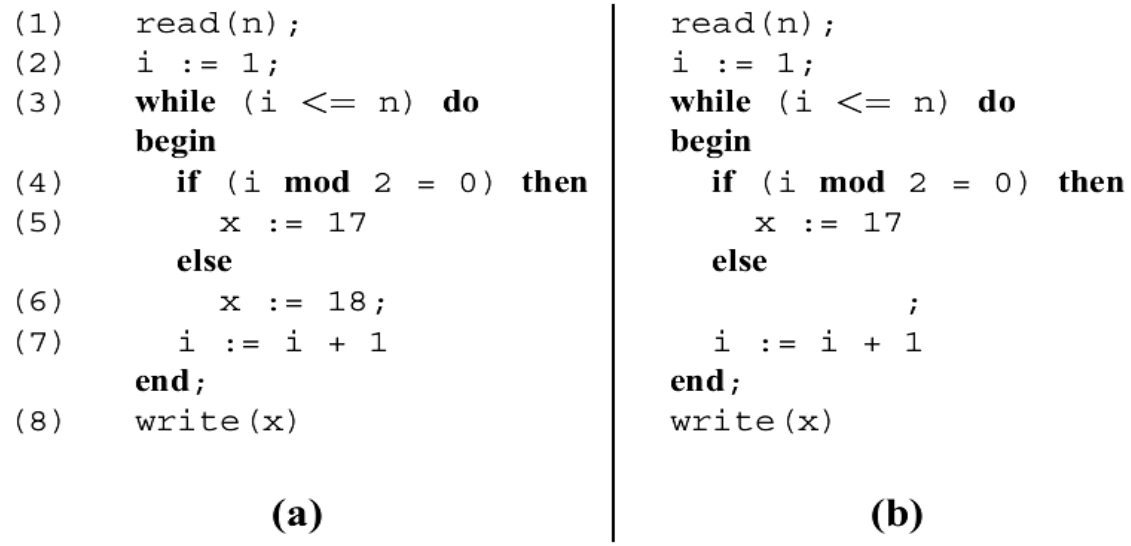
Static slicing có thể được sử dụng để xác định các bộ phận của chương trình có tiềm năng đóng góp vào sự tính toán của các chức năng được lựa chọn cho tất cả các chương trình đầu vào có thể. *Static slicing* là hữu ích để đạt được một sự hiểu biết chung của các bộ phận của chương trình đóng góp vào sự tính toán để các chức năng được lựa chọn. Mặc dù *static slicing* có nhiều lợi thế trong quá trình theo dõi chương trình, *static slicing* thường xuyên vẫn còn chương trình con lớn vì tính toán không chính xác của những *slice*. Ngoài ra *static slicing* không thể được sử dụng trong quá trình theo dõi về thực hiện chương trình.

1.3 Dynamic slicing

Khái niệm *dynamic slicing* ban đầu được giới thiệu bởi *Kroel* và *Laski* là một biến thể của phương pháp phân tích luồng ngược của *Balzer*. Phân tích luồng ngược quan tâm đến luồng thông tin trong chương trình với một giá trị đầu vào cụ thể. Người dùng duyệt đồ thị biểu diễn phụ thuộc điều khiển và phụ thuộc dữ liệu giữa các câu lệnh trong chương trình. Ví dụ như giá trị được tính tại câu lệnh *n* phụ thuộc vào giá trị tính toán tại câu lệnh *t* thì người dùng phải duyệt ngược từ đỉnh tương ứng với câu lệnh *n* đến đỉnh của câu lệnh *t*.

Lịch sử thực hiện là đường đi chứa một chuỗi các sự xuất hiện của các câu lệnh và tính chất điều khiển. Trong phương pháp *dynamic slicing* thì chỉ có các sự phụ thuộc xuất hiện trên một lịch sử thực hiện cụ thể của chương trình mới được cho vào *slice slicing*. Một tiêu chuẩn slice là bộ ba (x, I^q, V)trong đó x thể hiện đầu vào của chương trình, sự xuất hiện của câu lệnh I^q là thành phần thứ q trong đường đi, V là tập hợp các biến trong chương trình. Sự khác biệt giữa *static slicing* và

dynamic slicing là *dynamic slicing* sử dụng với giả thiết có đầu vào cố định cụ thể còn *static slicing* không quan tâm đến đầu vào.



```
(1)   read(n);
(2)   i := 1;
(3)   while (i <= n) do
      begin
(4)     if (i mod 2 = 0) then
(5)       x := 17
        else
(6)       x := 18;
(7)     i := i + 1
      end;
(8)   write(x)
```

(a)

```
read(n);
i := 1;
while (i <= n) do
begin
  if (i mod 2 = 0) then
    x := 17
  else
          ;
  i := i + 1
end;
write(x)
```

(b)

Hình 2: (a) chương trình mẫu, (b) *Dynamic slicing* với tiêu chuẩn (n=2, 8^1, x)

Ví dụ 1.3.1: Trong Hình 2 chỉ ra *slice dynamic slicing* của chương trình mẫu với tiêu chuẩn *slicing C = (n = 2, 8^1, x)* trong đó 8^1 thể hiện lần xuất hiện thứ nhất của câu lệnh *8* trong lịch sử thực hiện của chương trình. Với đầu vào *n = 2* thì thứ tự thực hiện các câu lệnh của chương trình là *{1^1, 2^2, 3^3, 4^4, 6^5, 7^6, 3^7, 4^8, 5^9, 7^{10}, 3^{11}, 8^{12}}*, các chỉ số bên trên chỉ thứ tự câu lệnh thực hiện(trong thứ tự này thì câu lệnh số *8* xuất hiện lần đầu ở vị trí thứ *12*). Với đầu vào *n = 2* thì vòng lặp thực hiện hai lần với các phép gán là *x*: = *18* và *x*: = *17*. Trong vòng lặp lần thứ hai thì câu lệnh gán *x*: = 17 được thực hiện thay thế cho phép gán trong câu lệnh *x*: = *18* thực hiện trong vòng lặp thứ nhất nên câu lệnh gán *x*: = *18* trong nhánh *else* của câu lệnh *if* không có trong *slice slicing*. Trong khi đó thì *slice static slicing* với tiêu chuẩn *C = (8, x)* bao gồm toàn bộ chương trình.

Chương 2: CÁC KĨ THUẬT DÙNG TRONG PHƯƠNG PHÁP STATIC SLICING

2.1.Static slicing đơn thủ tục.

2.1.1. Slicing dựa vào đồ thị luồng điều khiển

Phương pháp *static sling* dựa trên đồ thị luồng điều khiển được giới thiệu bởi *Weiser*. Một tiêu chuẩn *slice* trong phương pháp này là một bộ *C=(n, V)*trong đó *n* là một nút trong đồ thị luồng điều khiển *CFG* của chương trình và *V* là một tập con các biến của chương trình. Một *slice slicing S* đối với tiêu chuẩn *C=(n, V)* là một tập con các câu lệnh trong *P* thỏa mãn tính chất: khi *P* dừng với một giá trị đầu vào cho trước thì *S* cũng dừng với đầu vào đó, *P* và *S* cùng tính toán ra một giá trị của các biến trong *V* khi câu lệnh tương ứng với nút *n* được thực hiện.

Đồ thị luồng điều khiển *CFG* là một đồ thị có hướng với mỗi nút biểu diễn một câu lệnh hay tính chất điều khiển trong chương trình. Một cạnh từ nút *i* đến nút *j* thể hiện cho luồng điều khiển từ *i* đến *j*. Đồ thị *CFG* chứa hai nút đặc biệt là *Start* và *Stop* lần lượt thể hiện cho nút bắt đầu và kết thúc chương trình. Tại nút *i* tồn tại hai tập hợp đó là:

- Tập *DEF (i)* bao gồm tất cả các biến mà giá trị của nó bị thay đổi tại nút *i.*
- Tập *REF(i)* bao gồm tất cả các biến được tham chiếu tại nút *i.*

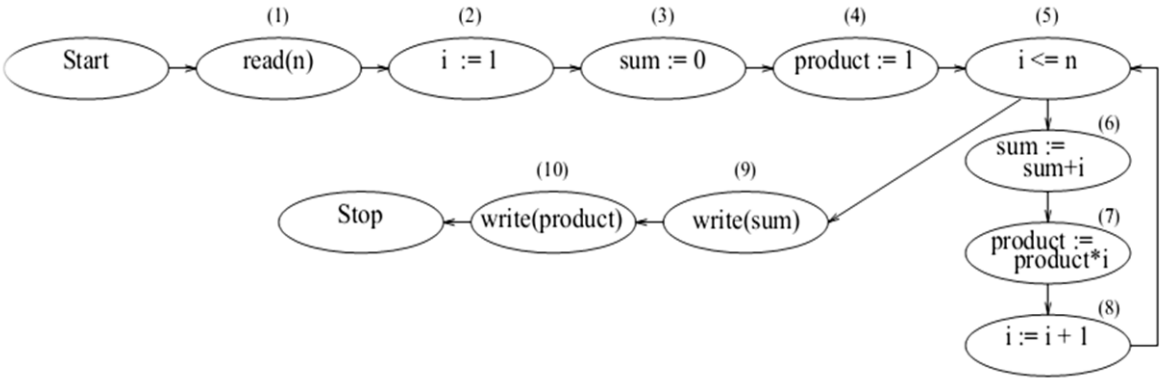
Trong đồ thị luồng điều khiển thì luồng điều khiển có hai loại là phụ thuộc dữ liệu và phụ thuộc điều khiển. Nút *j* là phụ thuộc luồng vào nút *i* nếu tồn tại một biến *x* sao cho:

- $x \in DEF(i)$
- $x \in REF(j)$
- Tồn tại một đường đi từ *i* đến *j* mà không có sự ghi dữ liệu lên biến *x*.

Một nút *i* trong đồ thị *CFG* là nút cha của một nút *j* nếu tất cả mọi đường đi từ nút *i* đến nút *Stop* đều phải qua nút *j*. Một nút *j* là phụ thuộc điều khiển vào nút *i* nếu:

- Tồn tại một đường đi *P* từ *i* đến *j* mà có $u \neq i, j$ trong *P* là cha của *j*
- *i* không phải là cha của *j*

Trong chương trình có cấu trúc thì các câu lệnh trong các nhánh của câu lệnh *if* và *while* là phụ thuộc điều khiển vào tính chất điều khiển.



Hình 3: Đồ thị *CFG* của chương trình mẫu trong Hình 1(a)

Ví dụ 2.1.1.1: Hình 3 chỉ ra *CFG* của chương trình trong Hình 1(a). Nút *7* là phụ thuộc luồng vào nút *4* vì nút *4* định nghĩa biến *product*, nút *7* tham chiếu biến *product* và tồn tại đường đi *4→5→6→7* không có định nghĩa biến *product*. Nút *7* là phụ thuộc điều khiển vào nút 5 vì tồn tại đường đi *5→6→7* có nút 6 là cha của nút *7* và nút *5* không là cha của nút *7*.

Slice slicing nhỏ nhất được tính bằng cách tính các tập biến liên quan ở từng nút trong đồ thị *CFG*. Đầu tiên, các biến liên quan trực tiếp được xác định bằng cách lấy ra các phụ thuộc dữ liệu. Ký hiệu $i \ _{CFG} j$ thể hiện cho tồn tại một cạnh trong *CFG* từ nút *i* đến nút *j*.

Với tiêu chuẩn cắt *C = (n, V)*, tập các biến liên quan trực tiếp tại nút *i* của *CFG* kí hiệu là *(i)*. Ta duyệt ngược đồ thị *CFG* để tìm ra các biến liên quan. Tập được xác định như sau:

- $R_C^0\ (i) = V$ khi $i=n$.
- Với mọi $i \rightarrow_{CFG} j$, $R_C^0\ (i)$ chứa tất cả các biến *v* sao cho $v \in R_C^0\ (j)$ và $v \notin DEF(i)$, hoặc $v \in REF(i)$ và $DEF(i) \cap R_C^0\ (j) \neq \varphi$.

Tập các câu lệnh liên quan trực tiếp ${S^{k+1}}_C$ là một tập các nút i xác định một biến *v* liên quan tại nút liền kề sau *j* và *i* trong *CFG*:

$$R_C^{k+1}(i) \quad = \quad R_C^k(i) \cup \bigcup_{b \in B_C^k} R^0_{(b,\mathrm{REF}(b))}(i)$$

Các biến được tham chiếu trong tính chất điều khiển của câu lệnh *if* hay *while* là liên quan gián tiếp nếu tồn tại ít nhất một câu lệnh trong thân của nó là liên quan.

Tầm ảnh hưởng *IFNL(b)* của câu lệnh nhánh *b* là liên quan gián tiếp nếu có *i* và *i* nằm trong tầm ảnh hưởng *IFNL(b)* của *b*. Các câu lệnh nhánh liên quan do ảnh hưởng của chúng lên nút *i* nằm trong được xác định như sau:

$$B_C^k \quad = \quad \{b \mid \exists i \in S_C^k,\ i \in \text{INFL}(b)\}$$

Tập các biến liên quan gián tiếp $S^{k+1}{}_C$ chứa các nút trong $B^k{}_C$ cùng với nút *i* xác định một biến liên quan đến một liền kề sau *j*:

$$S_C^{k+1} \quad = \quad B_C^k \cup \{i \mid \text{DEF}(i) \cap R_C^{k+1}(j) \neq \emptyset, i \rightarrow_{\text{CFG}} j\}$$

Các tập $R^{k+1}{}_C$ và $S^{k+1}{}_C$ là các tập không giảm lần lượt của các biến và câu lệnh trong chương trình. Quá trình tính toán trên được tiếp tục đến khi $S^{k+1}{}_C$ cố định. Các câu lệnh trong $S^{k+1}{}_C$ thu được là các câu lện có trong *slice slicing* mong muốn .

NODE #	DEF	REF	INFL	R_C^0	R_C^1
1	{ n }	∅	∅	∅	∅
2	{ i }	∅	∅	∅	{ n }
3	{ sum }	∅	∅	{ i }	{ i, n }
4	{ product }	∅	∅	{ i }	{ i, n }
5	∅	{ i, n }	{ 6, 7, 8 }	{ product, i }	{ product, i, n }
6	{ sum }	{ sum, i }	∅	{ product, i }	{ product, i, n }
7	{ product }	{ product, i }	∅	{ product, i }	{ product, i, n }
8	{ i }	{ i }	∅	{ product, i }	{ product, i, n }
9	∅	{ sum }	∅	{ product }	{ product }
10	∅	{ product }	∅	{ product }	{ product }

Hình 4: Kết quả của thuật toán của *Weiser* với chương trình trong Hình 2(a) và *slice slicing* với tiêu chuẩn C = (10, product)

Thuật toán *static program slicing* sử dụng đồ thị luồng dữ liệu của *Weiser* tính toán ra *slice static slicing* theo các bước sau:

Bước 1: Xác định các cạnh phụ thuộc luồng và phụ thuộc điều khiển để vẽ đồ thị luồng điều khiển cho chương trình.

Bước 2: Xác định các tập biến và các tập câu lệnh liên quan trực tiếp và gián tiếp đến câu lệnh và biến trong câu lệnh *slice* thông qua việc tính toán các tập hợp $R^0{}_C$, $S^0{}_C$, $R^{k+1}{}_C$, $S^{k+1}{}_C$

Bước 3: Lặp lại các bước tính toán các tập liên quan ở bước 2 trên để tính toán tập cố định $S^{k+1}{}_C$. Các câu lệnh trong $S^{k+1}{}_C$ là các câu lệnh có trong *slice program slicing* mong muốn.

Ví dụ 2.1.1.2: Xét chương trình trong Hình 3(a) với tiêu chuẩn *slice(10, product)*. Hình 4 chỉ ra các tập *DEF, REF, INFL* và tập các biến liên quan tính theo thuật toán của *Weiser*. Đồ thị *CFG* của chương trình chỉ ra trong Hình 3.Từ các thông tin trên hình vẽ ta tính được $S^0{}_C$ = {2, 4, 7, 8}, $B^0{}_C$ ={5}, $S^1{}_C$ ={1, 2, 4, 5, 7, 8}. Trong ví dụ này, tập cố định của tập các câu lệnh liên quan gián tiếp $S^1{}_C$ đạt được là tương ứng với *slice program slicing* trong hình1(b). Câu lệnh *write(product)* không có trong *slice slicing*. Thật ra câu lệnh xuất ra không có trong *slice slicing* vì: *(i) DEF* của nó rỗng nên không có câu lệnh nào phụ thuộc dữ liệu vào nó và *(ii)* không có câu lệnh nào phụ thuộc điều khiển vào một câu lệnh xuất.

2.1.2.Slicing dựa vào đồ thị phụ thuộc

Ottenstein đưa kĩ thuật *static program slicing* đơn thủ tục dựa trên đồ thị phụ thuộc chương trình. Các câu lệnh của chương trình tạo thành các đỉnh của đồ thị phụ thuộc, các cạnh tương ứng với phụ thuộc dữ liệu và phụ thuộc điều khiển giữa các câu lệnh.

Đồ thị *PDG* của chương trình *P* kí hiệu là G_P là một đồ thị có hướng mà các đỉnh của G_p biểu diễn các câu lệnh và tính chất điều khiển xuất hiện trong chương trình *P*, trong G_P có một đỉnh đặc biệt gọi là đỉnh vào. G_P là một đa đồ thị nên có thể có nhiều hơn một cạch giữa hai đỉnh.

Đồ thị G_P chứa một cạch phụ thuộc điều khiển từ đỉnh *u* đến đỉnh *v* kí hiệu là $u \rightarrow_c v$ nếu xảy ra một trong các điều sau đây:

- *u* là đỉnh vào và *v* biểu diễn cho một câu lệnh của *P* không lồng trong bất kỳ vòng lặp hay điều kiện nào. Các cạnh này được dán nhãn *True*.
- *u* biểu diễn cho một tính chất điều khiển và *v* biểu diễn một câu lệnh của *P* ở ngay trong vòng lặp hay điệu kiện biểu diễn bởi *u*. Nếu *u* là tính chất điều khiển của một vòng lặp *while* thì cạch $u \rightarrow_c v$ được gán nhãn *True*, nếu *u* là tính chất của một câu lệnh điệu kiện thì cạch $u \rightarrow_c v$ là được gán nhãn *True* hay *False* lần lượt tùy theo *v* xuất hiện trong nhánh *then* hay nhánh *else*.

Một cạnh phụ thuộc dữ liệu từ đỉnh *u* đến đỉnh *v* hàm ý chỉ ra sự tính toán của chương trình có thế bị thay đổi nếu thứ tự tương đối của các thành phần đại diện của *u* và *v* bị đảo ngược lại. Các cạn h phụ thuộc dữ liệu của *PDG* được xác định bằng cách sử dụng phân tích luồng dữ liệu. Một *PDG* chứa một cạnh phụ thuộc luồng từ đỉnh *u* đến đỉnh *v* ký hiệu là $u \rightarrow_f v$ nếu thỏa mãn các điều kiện sau đây:

1. *u* là đỉnh định nghĩa biến *x*.
2. *v* là một đỉnh sử dụng biến *x*.
3. Đường đi từ *u* đến *v* không có định nghĩa nào của *x*.

Các phụ thuộc luồng được chia thành hai loại là lặp mang và lặp độc lập. Một phụ thuộc luồng lặp mang $u \rightarrow_f v$ được truyền bởi vòng lặp *L*, ký hiệu là $u \rightarrow_{lc(L)} v$ nếu ngoài có 3 điều kiện (1), (2), (3) trên thì phải thỏa mãn thêm các điều kiện sau:

4. Có một đường đi thỏa mãn điều kiện thứ (3)bên trên và bao gồm một cạnh ngược đến tính chất điều khiển của vòng lặp *L*.
5. Cả hai *u* và *v* đều nằm trong vòng lặp *L*.

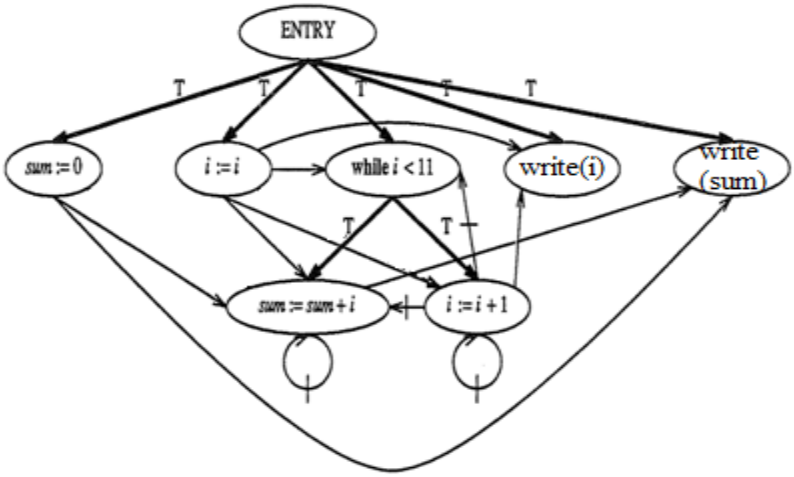
Một phụ thuộc luồng $u \rightarrow_f v$ là lặp độc lập kí hiệu là $u \rightarrow_{li} v$ ngoài các điều kiện (1), (2), (3) bên trên thì phải thỏa mãn thêm các điều kiện sau:

6. Có một đường đi thỏa mãn điều kiện thứ (3)ở trên và không có cạnh ngược đến tính chất điều khiểu của vòng lặp *L*.
7. Cả hai *u* và *v* đều nằm trong vòng lặp *L*.

Horwitz và *Reps* đã chỉ ra rằng các biến thể *PDG* là đầy đủ tức là nếu hai chương trình có cùng *PDG* thì chúng tương đương nhau. Điều đó có nghĩa là khi cho cùng một trạng thái đầu vào thì chúng tính ra cùng giá trị cho tất cả các biến hay chúng cùng chạy mà không kết thúc.

Ví dụ 2.1.2.1: Hình 5 chỉ ra một chương trình và biểu đồ phụ thuộc vào chương trình đó. Các mũi tên đậm thể hiện các cạnh phụ thuộc điều khiển, mũi tên nhạt thể hiện các cạnh phụ thuộc luồng lặp độc lập, mũi tên nhạt với dấu gạch ngang thể hiện các cạnh phụ thuộc luồng lặp mang.

```
Program Example
begin
        sum:= 0:
        i := 0;
        while(i<11)do
        begin
          sum := sum+1;
          i := i + 1;
        end
        write(sum);
        write(i);
end;
```



Hình 5: Ví dụ về đồ thị phụ thuộc chương trình

Tiêu chuẩn *slicing* đối với phương pháp *static slicing* sử dụng đồ thị phụ thuộc *PDG* chính là một đỉnh trong *PDG*. Ta kí hiệu các *slice slicing* của G_P đối với đỉnh *s* ký hiệu *slice(G, s). Slice(G, s)* chứa tất cả các đỉnh có thể đi tới đỉnh *s* thông qua các cạnh phụ thuộc luồng hay cạnh phụ thuộc điều khiển. Tập các đỉnh của *Slice(G,s)* kí hiệu là *V(Slice(G, s))* được xác định như sau:

$$V(Slice(G, s))=\{v \mid v \in V(G)\ v \rightarrow^*_{cf} s\}.$$

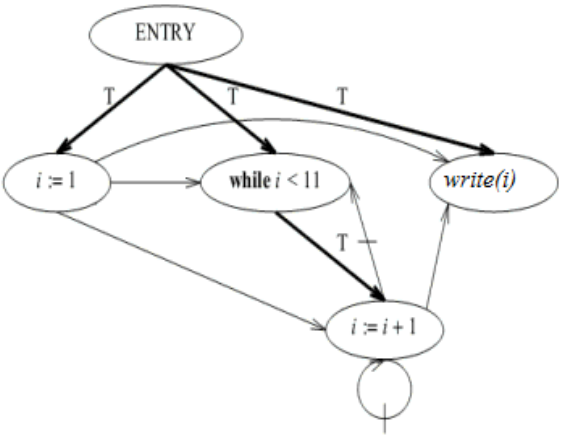
Tập các cạnh của *Slice(G, s)* kí hiệu là *E(Slice(G, s))* được xác định như sau:

$$E(Slice(G, s))=\{v \rightarrow_f u \mid (v \rightarrow_f u) \in E(G)\ v, u \in V(Slice(G, s))\}$$

$$\cup \{(v \rightarrow_c u) \in E(G)\ v, u \in V(Slice(G, s))\}.$$

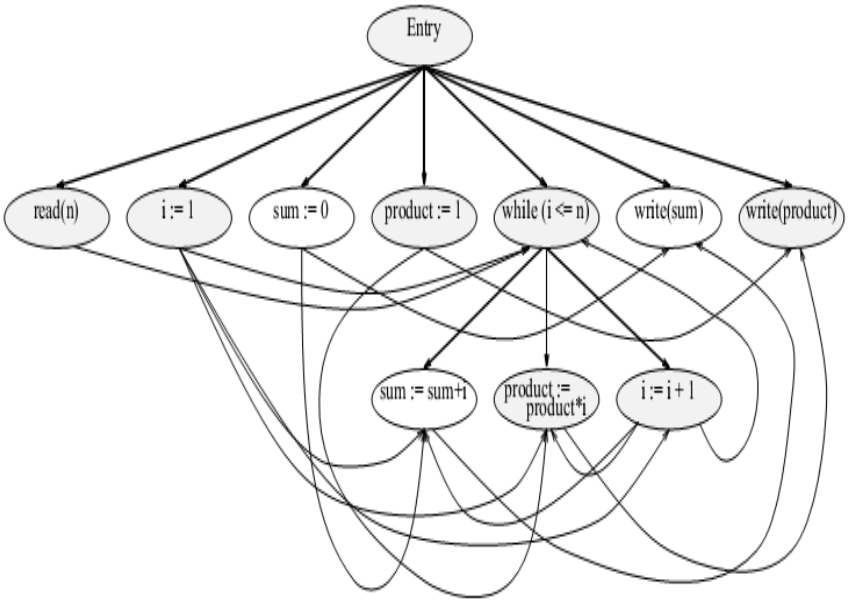
Ví dụ 2.1.2.2: Hình 6 chỉ ra *slice slicing* của chương trình theo đồ thị phụ thuộc của chương trình trong Hình 5 với tiêu chuẩn *slice write(i).*

```
Program Example
begin
        sum:= 0:
        i := 0;
        while(i<11)do
        begin
          sum := sum+1;
          i := i + 1;
        end
        write(sum);
        write(i);
end;
```



Hình 6: *Slice slicing* của chương trình trong Hình 5 với tiêu chuẩn *slicing write(i).*

Ví dụ 2.1.2.3: Trong Hình 7 chỉ ra đồ thị *PDG* của chương trình trong Hình 1(a). Trong hình này thì cạnh đậm biểu diễn phụ thuộc điều khiển cà cạnh nhạt biểu diễn phụ thuộc luồng. Các ô màu đậm chỉ ra các đỉnh trong slice slicing với tiêu chuẩn *write(product)*.



Hình 7: *PDG* của chương trình mẫu trong Hình 1(a)

2.2.static slicing đa thủ tục.

2.2.1.Slicing dựa theo đồ thị luồng điều khiển

Weiser chỉ ra hai bước để tính toán *slice program slicing* đa thủ tục dựa theo đồ thị luồng điều khiển như sau:

Bước một: trong chương trình có nhiều thủ tục thì *slice slicing* được tính toán với thủ tục *P* chứa tiêu chuẩn *slicing* ban đầu. Ảnh hưởng của lời gọi thủ tục lên tập các biến liên quan được tính bằng cách sử dụng thông tin tổng hợp đa thủ tục. Với một thủ tục *P*, thông tin này chứa tập *MOD(P)* của các biến bị thay đổi bởi *P* và tập *USE(P)* của các biến được sử dụng bởi *P*. Một lời gọi đến thủ tục *P* được xem như là nó xác định mọi biến trong *MOD(P)* và sử dụng mọi biến trong *USE(P)* nơi mà tham số thực thay thế cho tham số hình thức. Thuật toán của *Weiser* không chính xác do lấy các câu lệnh mà tham số ra phụ thuộc vào tham số vào.

Ví dụ 2.2.1.1: Trong Hình 8(a) chỉ ra một chương trình mẫu chỉ rõ vấn đề trên. Thuật toán *slicing* đa thủ tục sẽ tính ra *slice slicing* được thể hiện ở Hình 8(b) bằng cạch sử dụng các tập *MOD(P)* và *USE(P). Slice slicing* này chứa câu lệnh *a:=17* là do sự phụ thuộc giả giữa biến *a* trước lời gọi và biến *d* sau lời gọi.

```
     program Example;
     begin
(1)    a := 17;
(2)    b := 18;
(3)    P(a, b, c, d);
(4)    write(d)
     end

     procedure P(v, w, x, y);
(5)    x := v;
(6)    y := w
     end
```

(a)

```
program Example;
begin
  a := 17;
  b := 18;
  P(a, b, c, d);

end

procedure P(v, w, x, y);
        ;
  y := w
end
```

(b)

```
program Example;
begin
        ;
  b := 18;
  P(a, b, c, d);
  write(d)
end

procedure P(v, w, x, y);
        ;
  y := w
end
```

(c)

Hình 8: (a)Chương trình mẫu. (b)*Slice slicing* theo *weiser*. (c)*Slice slicing* theo *HRB*

Bước hai: Tiêu chuẩn *slicing* mới được sinh ra cho hai loại thủ tục: (i) những thủ tục *Q* được gọi bởi *P* và (ii) những thủ tục *R* gọi đến *P*.

Bước hai này sẽ được lặp lại đến khi không còn tiêu chuẩn mới được sinh ra. Việc sinh các tiêu chuẩn mới được hình thức hóa bằng công thức *UP(C)* và *DOWN(C)* ánh xạ một tập tiêu chuẩn *C* trong thủ tục *P* đến lần lượt các tập tiêu chuẩn trong các thủ tục *R* gọi *P* và trong các thủ tục *Q* được gọi bởi *P*.

Tập $UP(C) = \{(n_Q, v_Q)\}$ trong đó n_Q là câu lệnh cuối cùng của *Q* và v_Q là biến có liên quan trong *P* và trong phạm vi ảnh hưởng của *Q* (tham số hình thức thay thế cho tham số thực). Tập $DOWN(C) = \{(n_R, v_R)\}$ trong đó n_R là một lời gọi *P* trong *R* và v_R là tập biến liên quan trong câu lệnh đầu tiên của *P* trong tầm phạm vi ảnh hưởng của *R*(tham số thực thay thế cho tham số hình thức). Tập kín *(UP DOWN)*(C)* chứa các tiêu chuẩn *slice* cần thiết để tính toán một *slice slicing* với tiêu chuẩn *slicing* ban đầu C. Ta *slicing* chương trình theo tập tiêu chuẩn này sẽ được *slice program slicing* mong muốn

```
      program Example;
      begin
(1)   read(n);
(2)   i := 1;
(3)   sum := 0;
(4)   product := 1;
(5)   while i <= n do
      begin
(6)     Add(sum, i);
(7)     Multiply(product, i);
(8)     Add(i, 1)
      end;
(9)   write(sum);
(10)  write(product)
      end

      procedure Add(a; b);
      begin
(11)  a := a + b
      end

      procedure Multiply(c; d);
      begin
(12)  j := 1;
(13)  k := 0;
(14)  while j <= d do
      begin
(15)    Add(k, c);
(16)    Add(j, 1);
      end;
(17)  c := k
      end
```

Hình 9: Chương trình có cấu trúc đa thủ tục mẫu.

Ví dụ 2.2.1.2: *Slice slicing* được tính toán với giá trị cuối của biến *product* trong chương trình ở Hình 9. Giả sử ta phải *slicing* với tiêu chuẩn ban đầu là *(10, product).* Trong bước 1 của thuật toán *Weiser slice slicing* gồm tất cả các dòng của chương trình *Main* trừ dòng 3 và dòng 6. Lời gọi thủ tục Multiply*(product, i)* và Add*(i, 1)* cũng có trong *slice slicing* vì: (i)các biến *product* và *i* liên quan đến các lời gọi thủ tục đó , (ii) sử dụng phân tích luồng thông tin đa thủ tục ta có *MOD(Add)* = {a}, *USE(Add) = {a, b}, MOD(Multiply)* = {c}, *USE(Multiply) = {c, d}.* Với tiêu chuẩn ban đầu trong chương trình *Main*, ta có *UP*(*{10, product}*) = *DOWN*(*{10, product}*) chứa tiêu chuẩn *(11, {a})* và *(17, {c, d}).* Kết quả của việc cắt thủ tục *Add* với thủ tục*(11, {a})* và thủ tục *Multiply* với tiêu chuẩn *(17, {c, d})* chính là cả hai thủ tục đó. Lưu ý rằng các lời gọi *Add* ở dòng *15, 16* sinh ra tiêu chuẩn mới (*11, {a, b}*) và sau đó slicing lại thủ tục *Add*.

Horwitz, Reps và *Binkley* đã chỉ ra rằng phương pháp *slicing* đa thủ tục của *Weiser* không chính xác vì phương pháp này gặp vấn đề về ngữ cảnh gọi. Vấn đề ngữ cảnh gọi gặp phải khi tính toán xuống các thủ tục *Q* được gọi bởi thủ tục *P*. Nhưng khi duyệt ngược đồ thị luồng điều khiển thì sẽ đi lên mọi thủ tục gọi *Q* mà không chỉ có *P*. Điều này tương đương với các đường thực hiện vào *Q* từ *P* và ra *Q* từ một thủ tục *P'* khác. Các đường thực hiện đó là không thực thi được nên phương pháp *slicing* này sẽ tạo ra *slice slicing* không chính xác .

Ví dụ 2.2.1.3: Trong Hình 9 chỉ ra vấn đề ngữ cảnh gọi. Khi dòng 11 có trong *slice slicing*, tiêu chuẩn mới được sinh ra cho tất cả lời gọi đến *Add*. Các lời gọi đó là các lời gọi tại các dòng *8, 15, 16* và cũng có lời gọi *Add(sum, i)* ở dòng *6*. Tiêu chuẩn mới (*6*, {*sum, i*}) được sinh ra sẽ cho câu lệnh *3* và *6* vào trong slice slicing là cho *slice slicing* chương trình thu được toàn bộ chương trình.

Vấn đề ngữ cảnh gọi của thuật toán *Weiser* sẽ được giải quyết nếu các tiêu chuẩn trong tập *UP* chỉ được quan tâm khi có các thủ tục gọi đến thủ tục *P* chứa tiêu chuẩn khởi tạo. Khi không có thủ tục nào gọi đến thủ tục *P* thì chỉ có tập *DOWN* là cần thiết cho việc tính toán *slice slicing*.

```
Program Main;
...
  while (...)do
        P (x1, x2, ..., xn);
        z: = x1;
        x2: = x1;
        x3: = x2;
        ...
        xn: = x(n-1);
  end;
  write(z);
end;
```

```
Procedure P (y1, y2, ..., yn);
Begin
  write(y1);
  write(y2);
  ...

  write(yn);
end;
```

Hình 10: Chương trình mẫu mà thủ tục *P* bị *slicing n* lần với thuật toán của *Weiser*

Ví dụ 2.2.1.4: Trong Hình 10, *L* kí hiệu cho số dòng của câu lệnh *write(z)* và *M* là số dòng câu lệnh cuối của thủ tục *P*. Tính toán *slice slicing* với tiêu chuẩn (*L*, {*z*} yêu cầu *n* vòng lặp trong thân của vòng lặp *while*. Trong vòng lặp thứ *i* thì các biến x_1, ..., x_i sẽ liên quan tại đỉnh gọi gây ra bao gồm tiêu chuẩn (*m*, {y_1, ……, y_i}) trong *DOWN(Main)*. Nếu ta không chú ý khi lấy nhóm tiêu chuẩn trong *DOWN(Main)* thì thủ tục *P* sẽ bị *slicing n* lần.

2.2.2. Đồ thị phụ thuộc

Horwitz, Reps và *Binkley* sử dụng phương pháp *slicing* dựa vào đồ thị phụ thuộc hệ thống *SDG* để *slicing* chương trình đa thủ tục có cấu trúc. Từ "hệ thống" trong khái niệm đồ thị phụ thuộc hệ thống dùng để nhấn mạnh một chương trình có nhiều thủ tục. Một *SDG* chứa một *PDG* của chương trình chính và các đồ thị phụ thuộc thủ tục của mỗi thủ tục được kết nối bởi các cạnh phụ thuộc lường và cạnh phụ thuộc điều khiển giữa các thủ tục. Với mỗi câu lệnh gọi thì tương ứng có một đỉnh gọi trong *SDG*. Tham số truyền được biểu diễn bằng cách sử dụng bốn loại đỉnh tham số.

- Ở phương diện thủ tục gọi, tham số truyền được biểu diễn bằng đỉnh thực trong và đỉnh thực ngoài mô hình việc sao chép các biến thực lần lượt đến từ các biến tạm, được phụ thuộc điều khiển vào các đỉnh gọi
- Ở phương diện các thủ tục được gọi, tham số truyền được biểu diễn bằng đỉnh hình thức trong và hình thức ngoài mô hình việc sao chép các biến hình thức lần lượt đến từ các biến tạm, được phụ thuộc điều khiển vào đỉnh vào của thủ tục.

Tại các câu lệnh gọi thủ tục thì các tham số truyền theo tham trị được xây dựng theo các bước sau:

- Thủ tục gọi sao chép các tham số thực trong của nó đến các biến tạm trước khi gọi nó.
- Các tham số hình thức trong của thủ tục gọi được khởi tạo tương ứng với các biến tạm.
- Trước khi trở lại, thủ tục được gọi sao chép giá trị cuối cùng của tham số hình thức ngoài đến các biến tạm.
- Sau khi trở lại, thủ tục gọi cập nhật các tham số thực ngoài bằng cách sao chép các giá trị tương ứng với các biến tạm.

Đồ thị phụ thuộc thủ tục tạo thành một *SDG* bằng cách sử dụng ba loại cạnh mới sau:

1. Cạnh gọi là các cạnh phụ thuộc điều khiển giữa đỉnh gọi trong đỉnh gọi và đỉnh vào thủ tục tương ứng trong đồ thị phụ thuộc thủ tục.
2. Cạnh tham số trong giữa các đỉnh hình thức trong tại thủ tục được gọi và đỉnh thực trong tại đỉnh gọi
3. Cạnh tham số ngoài giữa các đỉnh hình thức ngoài tại thủ tục được gọi và đỉnh thực ngoài tại đỉnh gọi.

Cạnh gọi là một loại cạnh mới của cạnh phụ thuộc điều khiển. Cạnh tham số trong và cạnh tham số ngoài là hai loại cạnh mới của cạnh phụ thuộc dữ liệu. Thuận lợi của phương pháp tính này là các cạnh phụ thuộc luồng có thể được tính toán theo cách thông thường như cách phân tích luồng dữ liệu trong đồ thị luồng điều khiển của thủ tục. Đồ thị luồng điều khiển chứa các nút tương tự như các đỉnh thực trong, đỉnh thực ngoài, đỉnh hình thức trong, đỉnh hình thức ngoài của đồ thị phụ thuộc thủ tục. Một đồ thị luồng dữ liệu của thủ tục bắt đầu với một chuỗi các câu lệnh gán sao chép các giá trị từ các biến tạm gọi đến các tham số hình thức và kết thúc với một chuỗi các câu lệnh gán sao chép giá trị từ tham số hình thức trở lại các biến tạm. Mỗi câu lệnh gọi trong thủ tục được biểu diễn trong đồ thị luồng điều khiển bằng một chuỗi các câu lệnh gán sao chép giá trị từ tham số thực đến các biến

tạm gọi, tiếp theo là một chuỗi các câu lệnh gán sao chép giá trị trở lại các biến tạm từ các tham số thực.

```
Program Main;
begin
  sum: = 0;
  i: = 1;
  while(i < 11) do
  begin
        call A(sum, i);
  end;
  write(sum);
  write(i);
end;

Procedure A(x, y);
begin
  Call Add(x, y);
  Call Increment(y);
end;
```

```
Procedure Add(a, b);
begin
  a: = a+b;
end;


Procedure Increment(z);
begin
  call Add(z, 1);
end;
```

Hình 11: Chương trình có cấu trúc đa thủ tục mẫu khác

Gọi G_P là đồ thị phụ thuộc thủ tục của thủ tục *P*. Các đỉnh tham số liên kết với lời gọi từ thủ tục *P* đến thủ tục *Q* được xác định như sau:

- Trong G_P, các đỉnh bên dưới của đỉnh gọi trong đỉnh gọi biểu diễn lời gọi đến *Q*, có một đỉnh thực trong tương ứng với tham số thực *e* của lời gọi đến *Q*. Đỉnh thực trong được gán nhãn *r_in*: =*e* trong đó *r* là tên tham số hình thức tương ứng.
- Với mỗi tham số thực *a* là một biến có một đỉnh thực ngoại được gán nhãn *a*: = *r_out* trong đó *r* là tham số hình thức tương ứng.

Gọi G_Q là đồ thị phụ thuộc thủ tục của thủ tục *Q*. Các đỉnh tham số liên kết với đỉnh vào của thủ tục *Q* và kết thúc thủ tục *Q* được xác định như sau:

1. Với mỗi tham số hình thức *r* của *Q* thì G_Q chứa một đỉnh hình thức trong và một đỉnh hình thức ngoài. Các đỉnh đó được gán nhãn theo thứ tự là *r*: =*r_in* và *r_out*: =*r*.

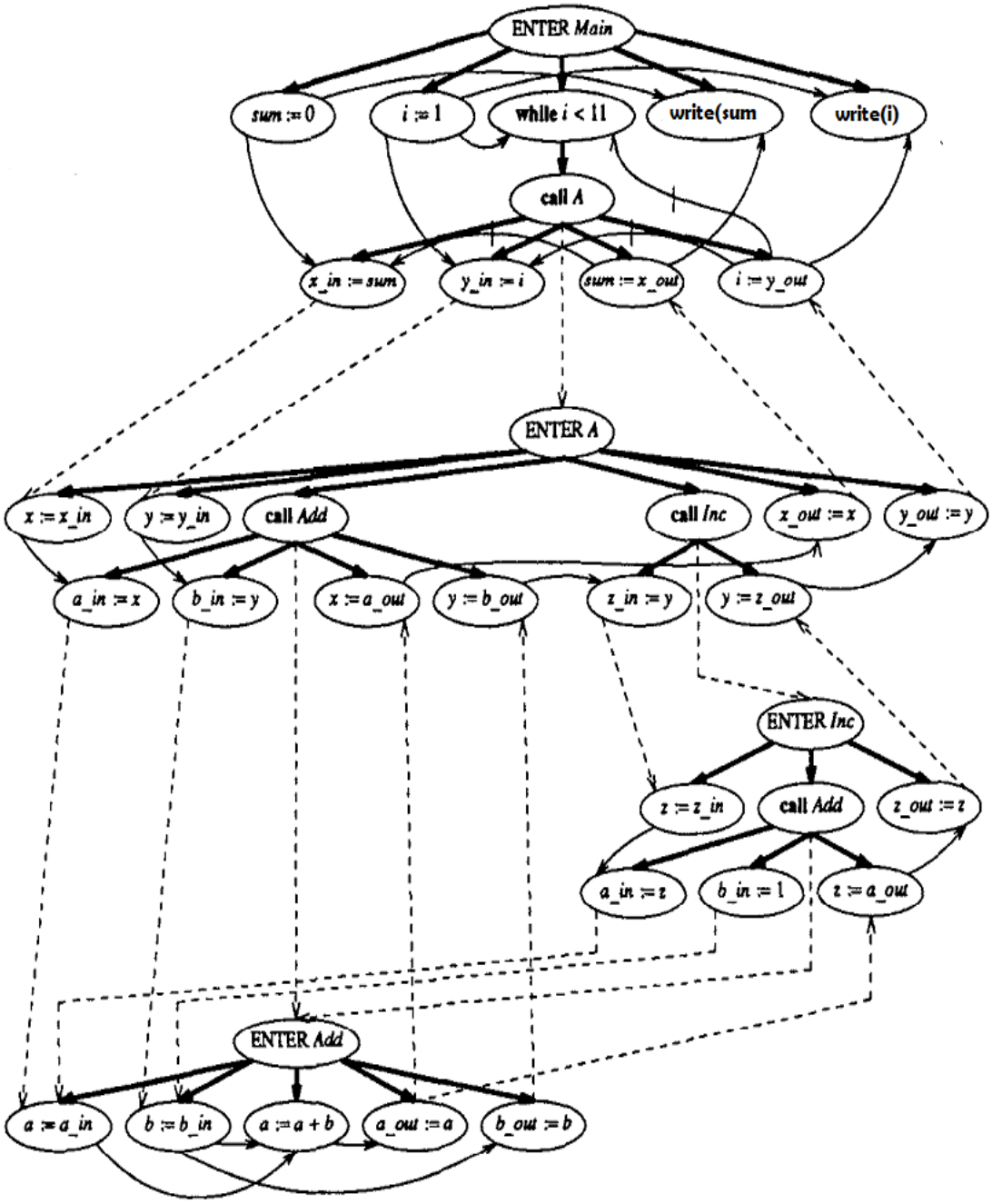
Ví dụ 2.2.2.1: Hình 12 thể hiện đồ thị phụ thuộc hệ thống của chương trình mẫu trong Hình 11. Đồ thị này gồm các đồ thị phụ thuộc thủ tục được kết nối với nhau thông qua các cạnh tham số trong, tham số ngoài, và cạnh gọi. Các cạnh điều khiển trong hình không có nhãn vì ta mặc định chúng có nhãn *True.*

Vấn đề ngữ cảnh gọi của phương pháp *static program slicing* đa cấu trúc của *Weiser* có thể được mnh họa đồ thị thể hiện trong Hình 12. Trong hình này có một đường đi từ đỉnh của thủ tục *Main* gán nhãn *i*: =*y*_out mặc dù giá trị của *i* sau lời gọi đến thủ tục *A* độc lập vơi giá trị của *sum* trước khi gọi. Đường đi cụ thể như sau:

Main: *x_in*: =*sum*

→ *A: x: = x_in*
→ *A: a_in: =x*
→ *Add: a: = a_in*
→ *Add: a: =a+b*
→ *Inc: z: =a_out*
→ *Inc: z_out: = z*
→ *A: y: = z_out*
→ *A: y_out: = y*
→ *Main: i: =y_out.*

Hình 12: Đồ thì *SDG* của chương trình đa thủ tục mẫu trong Hình 11



Nguyên nhân gây ra vấn đề này là không phải tất cả các đường đi trong đồ thị tương ứng với đường đi có thể thực thi. Ví dụ như đường đi từ đỉnh *x_in*: = *sum*

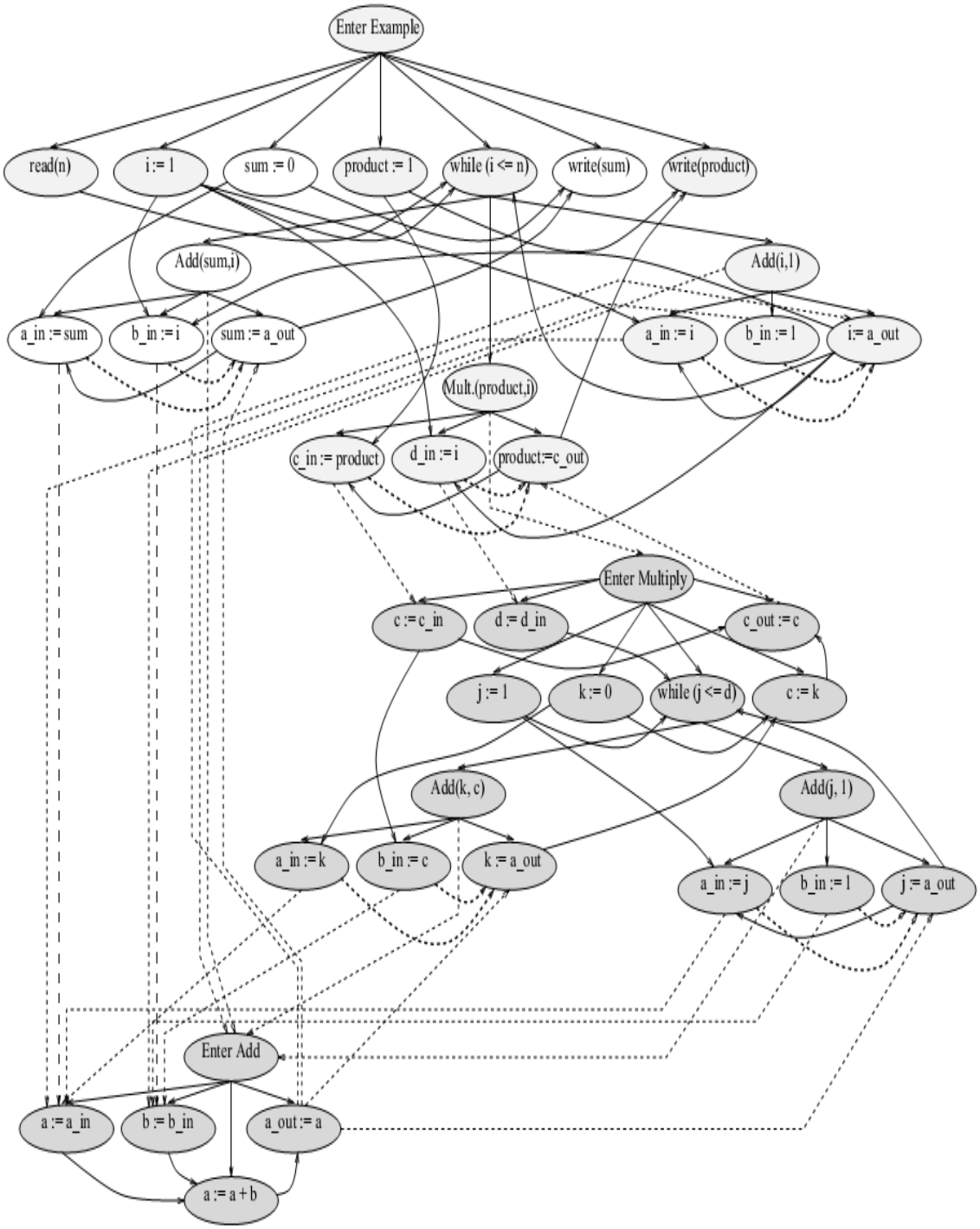
của thủ tục *Main* đến đỉnh *i*: *=y_out* của Main tương ứng với thủ tục *Add* được gọi bởi thủ tục A, nhưng trở lại thủ tục *Increment*. Để khắc phục vấn đề này, chúng ta thêm một loại cạnh bổ sung vào đồ thị phụ thuộc hệ thống biểu diễn phụ thuộc ngoài do ảnh hưởng của các lời gọi thủ tục. Ví dụ, đối với đồ thị thể hiện trong Hình 12, có cạnh phụ thuộc ngoài từ đỉnh *x_in*: = *sum* của thủ tục *Main* đến đỉnh *sum*: = *x_out* của *Main*. Sự phụ thuộc này tồn tại vì giá trị của biến *sum* sau khi gọi đến A phụ thuộc vào giá trị của *sum* trước khi các cuộc gọi đến *A*.

Giả sử ta phải tính toán *slice slicing* chương trình theo đỉnh *s* nằm trong thủ tục *P*. Phương pháp *static slicing* đa thủ tục sử dụng đồ thị phụ thuộc hệ thống tính toán *slice slicing* bằng cách duyệt đồ thị *SDG* theo hai bước:

Bước một: Xác định các đỉnh trong thủ tục *P* mà *s* đi tới nhưng chưa đi vào trong các thủ tục *Q* được gọi bởi *P*. Các cạnh phụ thuộc đa thủ tục ngoài chỉ từ đỉnh gọi đến đỉnh vào của thủ tục nhưng chưa thật sự đi vào thủ tục được gọi.

Bước hai: Thuật toán cắt tiếp tục với các đỉnh vào của các thủ tục được gọi đã chỉ ra ở bước một và xác định các đỉnh còn lại trong *slice slicing*.

Ví dụ 2.2.2.2: Hình 13 chỉ ra đồ thị *SDG* cho chương trình đa thủ tục trong Hình 9. Trong hình này, phân tích luồng thông tin đa thủ tục được sử dụng để loại bỏ các đỉnh của tham số thứ hai của thủ tục *Add* và *Multiply*. Trong hình thì cạch nhạt thể hiện phụ thuộc luồng, cạnh đậm thể hiện phụ thuộc điều khiển, nét đứt nhạt thể hiện các lời gọi, phụ thuộc tham số trong và phụ thuộc tham số ngoài, nét đứt đậm thể hiện phụ thuộc luồng đa thủ tục ngoài.



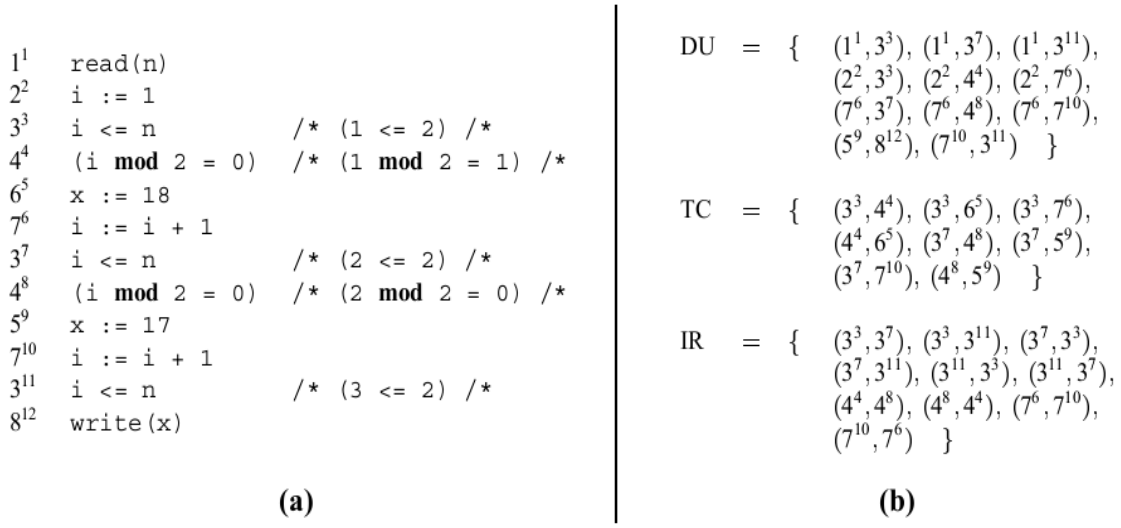
Hình2.2.2. 1: *SDG* của chương trình mẫu trong Hình 2.2.1.2

Chương 3: CÁC KỸ THUẬT DÙNG TRONG PHƯƠNG PHÁP DYNAMIC SLICING

3.1: *Phương thức dynamic chương trình đơn thủ tục*

3.1.1: *Các khái niệm luồng động*

Trong phương pháp *dynamic slicing* chương trình có cấu trúc đơn thủ tục, *Korel* và *Laski* coi lịch sử thực hiện chương trình là một đường đi chứa một chuỗi các sự xuất hiện của các câu lệnh và tính chất điều khiển. Các nhãn bên trên được sử dụng để phân biệt các lần xảy ra khác nhau của một câu lệnh trong lịch sử thực hiện. Giả sử có lịch sử thực hiện sau *{ 1^1, 2^2, 3^3, 4^4, 7^5, 4^6, 8^7}* thì câu lệnh *4* xuất hiện hai lần ở vị trí thứ *4* và thứ *6* trong lịch sử thực hiện.



Hình 14: (a)Đường đi của chương trình mẫu trong Hình 2(a). (b)các khái niệm luồng động cho đường đi đó.

Ví dụ 3.1.1.1: Hình 14 thể hiện đường đi của chương trình trong Hình 2(a) với đầu vào *n = 2*. Đường đi của lịch sử thực hiện của chương trình là*{1^1, 2^2, 3^3, 4^4, 6^5, 7^6, 3^7, 4^8, 5^9, 7^{10}, 3^{11}, 8^{12}}*.

Một tiêu chuẩn *dynamic slicing* trong kĩ thuật slicing này là một bộ ba (x, I^q, V)trong đó *x* thể hiện đầu vào của chương trình, sự xuất hiện của câu lệnh I^q là thành phần thứ *q* trong lịch sử thực hiện. *V* là tập con các biến trong chương trình. Các *slice dynamic slicing* phải thỏa mãn các yêu cầu sau: *(i)* câu lệnh tương ứng với tiêu chuẩn I^q phải có trong *slice slicing*, và *(ii)* nếu một vòng lặp xảy ra trong *slice slicing* thì nó được thực hiện cùng số lần nó được thực hiện trong chương trình ban đầu.

Korel và *Laski* giới thiệu ba khái niệm luồng động hình thức hóa sự phụ thuộc giữa các lần xảy ra của các câu lệnh trong đường đi:

1. Quan hệ định nghĩa, sử dụng (*DU)*: là một liên kết sử dụng của một biến với định nghĩa cuối cùng của nó.
2. Quan hệ kiểm tra, điều khiển (*TC)*:là liên kết giữa sự xuất hiện của tính chất điều khiển và câu lệnh, xảy ra trên đường đi phụ thuộc điều khiển vào nó, quan hệ này chỉ có trong kiểu hướng cú pháp chỉ xây dựng cho chương trình có cấu trúc.
3. Quan hệ định danh (*IR)*: là liên kết giữa sự xuất hiện khác nhau của cùng một câu lệnh.

Ví dụ 3.1.1.2: Hình 14(b) chỉ ra các khái niệm luồng dynamic cho đường đi trong Hình 14(a).

Các *slice dynamic slicing* được tính toán bằng cách xác định liên tiếp tập S^i của các câu lệnh liên quan trực tiếp và gián tiếp. Với tiêu chuẩn *(x, I^q, V)* thì phép tính khởi tạo S^0 chứa định nghĩa cuối cùng của các biến trong *V* trên đường đi cũng như kiểm tra các hành động trên đường đi mà ở đó I^q phụ thuộc điều khiển. Tính toán S^{i+1} được xác định như sau:

$$S^{i+1} = S^i \cup A^{i+1}$$

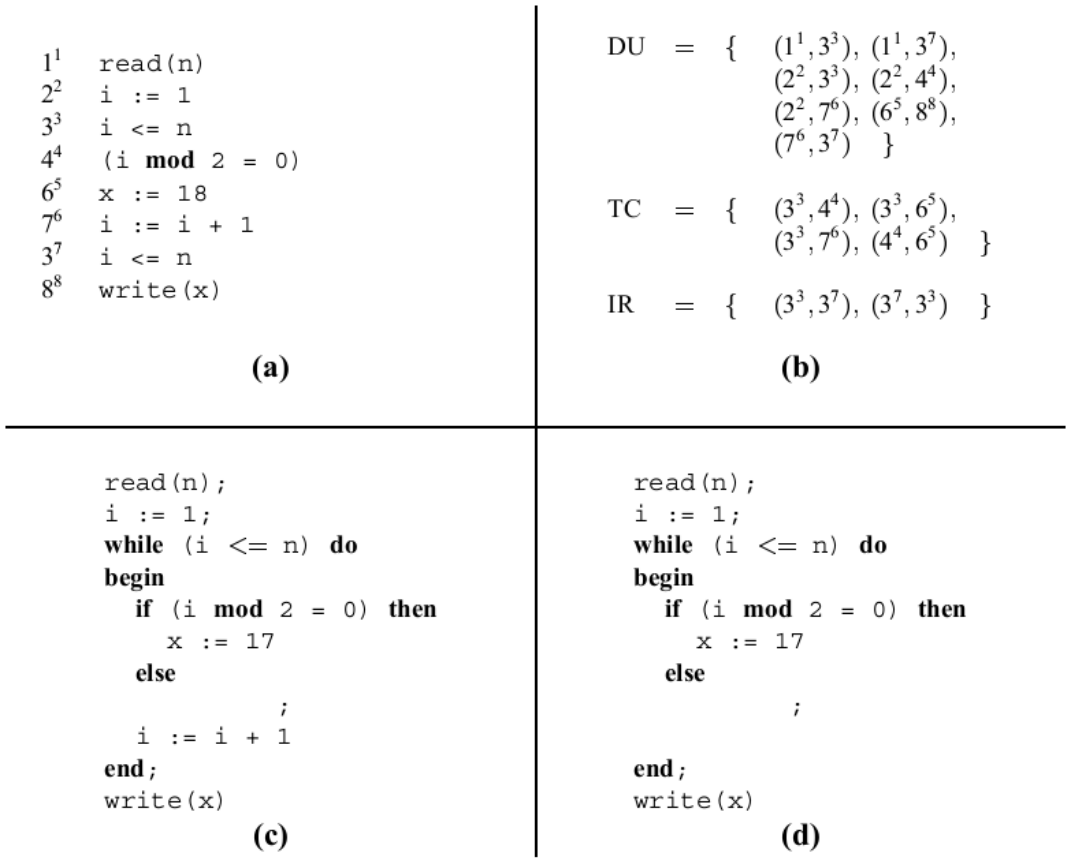
$$A^{i+1} = \{\, X^p \mid X^p \notin S^i,\ (X^p, Y^t) \in (\mathrm{DU} \cup \mathrm{TC} \cup \mathrm{IR}) \text{ for some } Y^t \in S^i,\ p < q \,\}$$

Slice dynamic slicing thu được dễ dàng từ tập cố định S_C của quá trình trên. *Slice slicing* gồm tất cả các câu lệnh *X* mà có thể thực hiện X^p xuất hiện trong S_C và câu lệnh *i* tương ứng với tiêu chuẩn I^q.

Ví dụ 3.1.1.3: Ta tính *slice dynamic slicing* cho đường đi trong Hình 14 với tiêu chuẩn *(n = 2, 8^{12}, {x}).* Lúc đầu tập S^0 chứa định nghĩa cuối cùng của *x* là $S^0 = \{5^9\}$. Tiếp theo ta sẽ tính được $A^1 = \{3^7, 4^8\}$, $A^2 = \{7^6, 1^1, 3^3, 3^{11}, 4^4\}$ và $A^3 = \{2^7, 7^{10}\}$. Từ đó theo quá trình trên ta tính được:

$S_C = \{1^1, 2^2, 3^3, 4^4, 7^6, 3^7, 4^8, 5^9, 7^{10}, 3^{11}, 8^{12}\}$

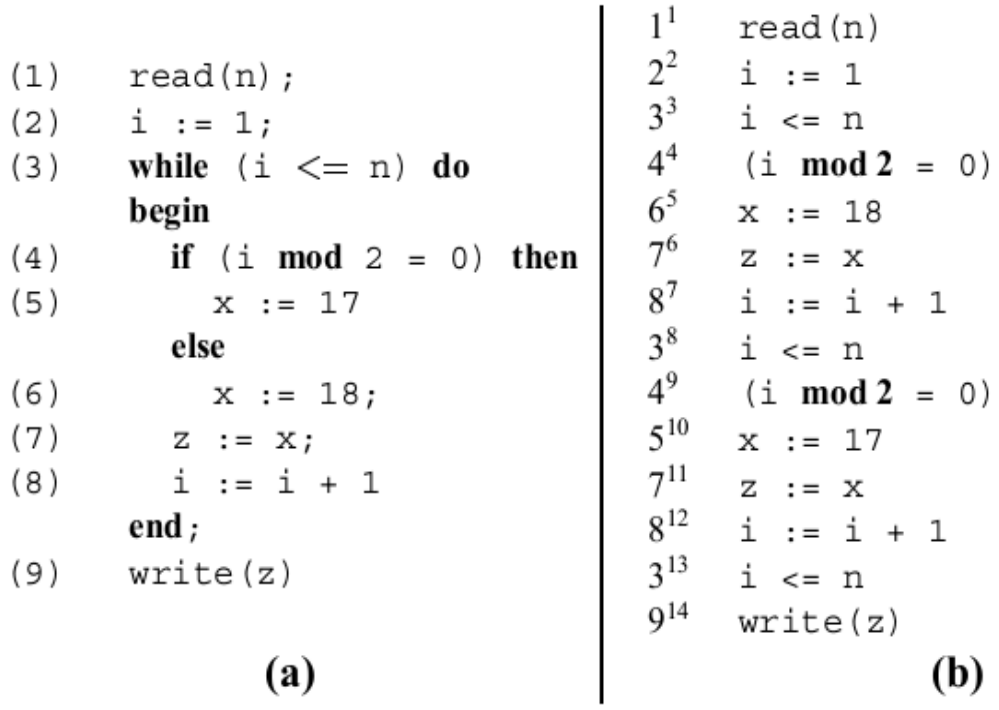
Vì vậy *slice dynamic slicing* với tiêu chuẩn *(n= 2, 8^{12}, {x})* chứa mọi câu lệnh của chương trình trừ câu *5* tương ứng với câu 6^5 trong đường đi. *Slice slicing* của chương trình này được chỉ ra trong Hình 2(b).



Hình 15: (a) Đường đi của chương trình mẫu trong Hình 2(a) với đầu vào *n= 1*. (b) Các khái niệm luồng động cho đường đi này. (c) *Slice dynamic slicing* với tiêu chuẩn *(n = 1, 8^8, x)*. (d*) Slice slicing* không dừng thu được nếu bỏ qua quan hệ *IR*

Vai trò của quan hệ định danh *IR* là duyệt chương trình theo cả hai hướng và cho vào tất cả các câu lệnh và tính chất điều khiển cần thiết để đảm bảo kết thúc các vòng lặp trong *slice slicing*.

Ví dụ 3.1.1.4: Chúng ta xem xét vai trò của quan hệ định danh *IR*. Xét đường đi của chương trình mẫu trong Hình 2(a) với đầu vào *n = 1* thể hiện trong Hình 15(a). Các khái niệm luồng *dynamic* với đường đi đó và *slice slicing* với tiêu chuẩn *(n= 1, 8^8, {x})* được thể hiện trong Hình 15(b) và *15*(c). *Slice slicing* thu được là một chương trình kết thúc. Tuy nhiên nếu ta tính các *slice slicing* không sử dụng quan hệ định danh *IR* thì sẽ thu được một chương trình không kết thúc thể hiện trong Hình 15(d). Lý do của hiện tường này là quan hệ *DU* và *TC* chỉ duyệt đường đi theo hướng ngược lại.



Hình 16: (a) Chương trình mẫu. (b)Đường đi với đầu vào n = 2.

Tuy nhiên duyệt quan hệ *IR* theo hướng ngược lại làm cho *slice slicing* chứa các câu lệnh không cần thiết để đảm bảo sự kết thúc.

Ví dụ 3.1.1.5: Hình 16(a) chỉ ra một phiên bản sửa đổi của chương trình trong Hình 2(a). Hình 16(b) thể hiện đường đi của chương trình đó. Từ đường đi đó ta có*(*7^6*,* 7^{11}*)*phụ thuộc *IR, (*6^5*,* 7^6*)* thuộc *DU* và *(*5^{10}*,* 7^{11}*)* thuộc *TC*. Do đó cả câu lệnh *5* và *6* đều có trong slice slicing mặc dù câu lệnh *6* không cần thiết để tính giá trị cuối cùng của *z* hay để kết thúc vòng lặp.

3.1.2.Đồ thị phụ thuộc

Agrawal và *Horgan* phát triển một hướng tiếp cận sử dụng đồ thị phụ thuộc để tính toán *slice dynamic slicing*. Thuật toán đầu tiên của họ tính toán *slice dynamic slicing* không chính xác nhưng nó có ích để hiểu các thuật toán sau này.

Hướng tiếp cận đầu tiên sử dụng đồ thị *PDG* đánh dấu các đỉnh là "đã thực hiện" cho tập các đỉnh được đưa ra. Một *slice dynamic slicing* được tính toán thông qua cách tính một *slice sitatic slicing* cho đồ thì con chỉ gồm các đỉnh được đánh dấu của *PDG*. Giải pháp này không chính xác vì nó không tính đến tình huống tồn tại một cạnh luồng trong *PDG* giữa hai đỉnh được đánh dáu v_1 và v_2 nhưng có định nghĩa của v_1 không được sử dụng tại v_2. Mặt khác, khi ta biểu diễn một đỉnh được

đánh dấu trong nhiều vòng lặp sẽ được biểu diễn trong tất cả các vòng lặp tiếp theo thậm chí khi các sự phụ thuộc không được lặp lại.

Ví dụ 3.1.2.1: Hình 18(a) chỉ ra đồ thì *PDG* của chương trình mẫu ở Hình 2(a). Giả sử ta muốn tính một *slice dynamic slicing* với giá trị cuối cùng của *x* cho đầu vào *n =2*. Tất cả các đỉnh của *PDG* được thực hiện vì mọi đỉnh của *PDG* đều được đánh dấu. Thuật toán *static slicing* ở phần 2.1.2 sẽ tính ra *slice dynamic slicing* là toàn bộ chương trình mẫu. Tuy nhiên, ta nhận thấy câu lệnh gán *x*: *= 18* là không liên quan. Phép gán này được cho vào *slice slicing* vì tồn tại một cạnh phụ thuộc luồng từ đỉnh *x = 18* đến đỉnh *write(x)* nhưng không biểu diễn một sự phụ thuộc trong vòng lặp thứ hai. Chính xác hơn thì sự phụ thuộc này chỉ xuất hiện trong vòng lặp khi biến điều khiển *i* có giá trị lẻ.

Giải pháp thứ hai cũng dựa trên đồ thì *PDG* có các đỉnh phân biệt trong đồ thì phụ thuộc tương ứng với mỗi lần xuất hiện của câu lệnh trong đường đi. Loại đồ thị này gọi là đồ thị phụ thuộc động *DDG*.Một tiêu chuẩn *dynamic slicing* được xác định là một đỉnh trong *DDG* và một *slice dynamic slicing* được tính toán bằng cách lấy tất cả các đỉnh *DDG* mà từ đỉnh tiêu chuẩn có thể đi tới. Một câu lệnh hay tính chất điều khiển có trong *slice slicing* nếu tiêu chuẩn được xem xét từ ít nhất một đỉnh trong sự xuất hiện của nó.

Nhược điểm của đồ thị *PDG* là số đỉnh bằng số câu lệnh thực thi không bị ràng buộc.

Ví dụ 3.1.2.2: Hình 18(c) chỉ ra đồ thị DDG cho chương trình mẫu ở Hình 16(a). Tiêu chuẩn *slicing* tương ứng với đỉnh có nhãn *write(z)* và mọi đỉnh từ đỉnh này có thể đi tới được in đậm.Ta thấy là tiêu chuẩn không được xem xét từ đỉnh có nhãn *x; = 18* cho nên câu lệnh gán tương ứng không có trong *slice slicing*.

```
program Q^n;
  read(x_1);
  ...
  read(x_n);
  MoreSubsets := true;
  while MoreSubsets do
  begin
    Finished := false;
    y := 0;
    while not(Finished) do
    begin
      read(i);
      case (i) of
        1:  y := y + x_i;
        ...
        n:  y := y + x_n;
      end;
      read(Finished);
    end;
    write(y);
    read(MoreSubsets);
  end
end.
```

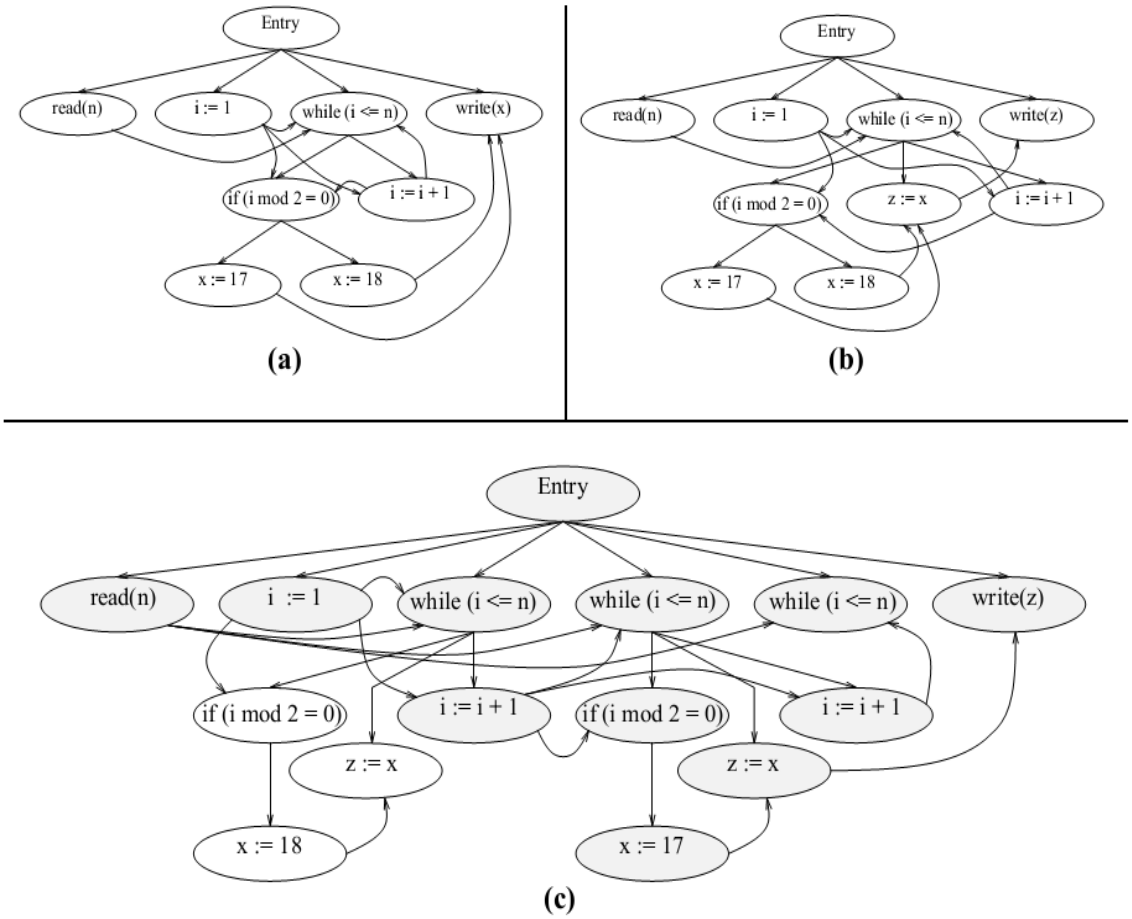
Hình 17: Chương trình Q^n có $O(2^n)$ *slice dynamic slicing* khác nhau

Trong giải pháp thứ ba, *Agarwal* và *Horgan* đề xuất cách giảm số đỉnh trong *DDG* bằng cách kết hợp các đỉnh có phụ thuộc ngoài ánh xạ đến cùng một tập các lệnh. Nói theo cách khác, một đỉnh mới chỉ được xem xét nếu nó tạo ra *slice dynamic slicing* mới. Rõ ràng thì cách kiểm tra này làm tăng số lần thực thi. Đồ thị kết quả được gọi là đồ thị phụ thuộc *dynamic* rút gọn *RDDG* của một chương trình. *Slice dynamic slicing* tính toán sự dụng *RDDG* có cùng độ chính xác như khi tính bằng *DDG*.

Thuật toán dùng đồ thị *RDDG* không có tất cả đường đi thực hiện. Trong đồ thị này cần phải lưu ý những vấn đề sau:

- Với mỗi biến, đỉnh tương ứng đến định nghĩa cuối cùng của nó.
- Với mỗi tính chất, các đỉnh tương ứng đến thực hiện cuối cùng của nó.
- Với mỗi đỉnh trong *RDDG* thì các *slice dynamic* chứa đỉnh đó.

Ví dụ 3.1.2.3: Trong đồ thị *DDG* ở Hình 18(c) thì đỉnh có nhãn *i = i+1* và hai đỉnh bên phải có nhãn *while (i<=n)* là *read(n)* và *i*: *= 1* có cùng phụ thuộc ngoài *dynamic*. Các đỉnh phụ thuộc vào các câu lệnh 1, 2, 3 và 8 của chương trình thể hiện ở Hình 16(a). Đồ thị *RDDG* cho chương trình này với đầu vào *n = 2* thu được bằng cách gộp bốn đỉnh bên trong đồ thị *DDG* thành một đỉnh.



Hình 18: (a) *PDG* của chương trình trong Hình 2(a).(b)*PDG* của chương trình trong Hình 16(a).(c)*DDG* của chương trình Hình 16(a).

3.2. Dynamic slicing đa thủ tục

Agrawal, Demillo và *Spafford* xem xét phương pháp *dynamic slicing* đa thủ tục với các cơ chế truyền tham số khác nhau. Giả sử thủ tục *P* với các tham số hình thức f_1, ..., f_n được gọi bởi tham số thực a_1, ..., a_n . Điều chú ý trong hương tiếp cận này là phụ thuộc dữ liệu động dựa trên các định nghĩa và các sử dụng của vùng nhớ đã sử dụng. Theo cách này ta tránh được hai bấn đề: *(i)*cách sử dụng biến toàn cục trong các thủ tục không gây ra lỗi, *(ii)* không có yêu cầu phân tích các bí danh.

Tham số truyền tham trị được tạo ra bằng một chuỗi các câu lệnh gán f_1: = a_1, ..., f_n: = a_n được thực hiện trước khi vào thủ tục. Để xác định các ô nhớ cho việc ghi hành động chính xác thì tập *USE* cho các tham số thực a_i được xác định trước khi vào thủ tục, tập *DEF* cho các tham số hình thức f_i được xác định sau khi vào thủ tục. Với tham số truyền tham trị kết quả, các phép gán của tham số hình thức đến tham số thực phải được thực hiện khi ra khỏi thủ tục. Tham số truyền tham chiếu không yêu cầu các chi tiết hành động đến *slice dynamic slicing* như cùng ô nhớ liên kết tương ứng giữa tham số thực và tham số hình thức a_i và f_i.

Một hướng tiếp cận khác của *dynamic slicing* đa thủ tục được giới thiệu bởi *Kamkar, Shahmehri* và *Fritzson* trong nghiên cứu các xác định tập các đỉnh gọi trong chương trình ảnh hưởng đến giá trị của một biến tại một đỉnh gọi cụ thể. Trong khi thực hiện, một đồ thị tổng hợp phụ thuộc động *DDSG* được xây dựng, các đỉnh trong đồ thị chỉ ra các thể hiện thủ tục tương ứng với sự kích hoạt thủ tục được chú thích bằng các tham số của nó. Các cạnh của đồ thị tổng hợp tương ứng với lời gọi thủ tục hay các cạnh phụ thuộc tổng hợp.

Một tiêu chuẩn *slicing* được xác định là một cặp chứa một thể hiện thủ tục và một tham số ra hoặc vào của thủ tục liên kết. Sau khi xây dựng đồ thị tổng hợp, một *slice slicing* với tiêu chuẩn *slicing* được xác định qua hai bước. Đầu tiên các phần của đồ thị tổng hợp có thể đi tới đỉnh tiêu chuẩn được xác định, đồ thị con được sinh ra là một *slice slicing* thực thi. Các đỉnh của *slice slicing* thực thi là một thể hiện thủ tục cụ thể vì các tham số có thể bị *slicing* đi. Một *slice slicing* chương trình đa thủ tục chứa tất cả các đỉnh gọi trong chương trình mà có một thể hiện cụ thể xuất hiện trong *slicing* thực thi.

Có ba phương pháp để ta có thể xây dựng một đồ thị tổng hợp. Trong phương pháp tiếp cận đầu tiên, các phụ thuộc dữ liệu đơn thủ tục được xác định một cách *static* tạo *ra slice slicing* không chính xác nếu có các câu lệnh điều kiện. Trong hướng tiếp cận thứ hai, tất cả các phụ thuộc được xác định lúc thực thi. Muốn có *slice slicing* chính xác thì các phụ thuộc của một thủ tục *P* phải được tính mỗi khi *P* được gọi. Cách tiếp cận thứ ba kết hợp sự hiệu quả của hướng tiếp cận static thứ nhất cùng sự chính xác của hướng tiếp cận *dynamic* thứ hai bằng cách tính toán các phụ thuộc bên trong các khối cơ bản *static* và các phụ thuộc *dynamic* giữa các khối. Trong mọi cách tiếp cận trên thì các phụ thuộc điều khiển được xác định *static*.

Chương 4: THỰC NGHIỆM TRÊN CÁC CHƯƠNG TRÌNH SLICER

Trong chương này em xin trình bày về hai chương trình được sử dụng minh họa cho hai phương pháp StaticSlicing và Dynamic Slicing.

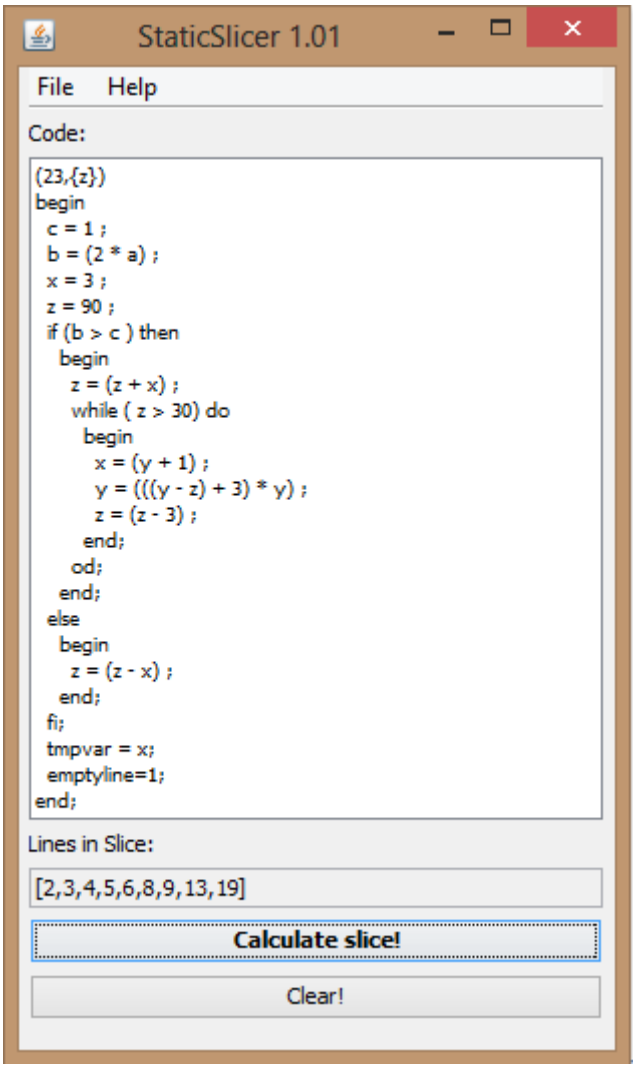
4.1 Chương trình StaticSlicer

Bài toán cho chương trình như sau:

```
(1)begin
(2)  c = 1 ;
(3)  b = (2 * a) ;
(4)  x = 3 ;
(5)  z = 90 ;
(6)  if (b > c ) then
(7)    begin
(8)      z = (z + x) ;
(9)      while ( z > 30) do
(10)        begin
(11)          x = (y + 1) ;
(12)          y = (((y - z) + 3) * y) ;
(13)          z = (z - 3) ;
(14)        end;
(15)      od;
(16)    end;
(17)  else
(18)    begin
(19)      z = (z - x) ;
(20)    end;
(21)  fi;
(22)  tmpvar = x;
(23)  emptyline=1;
(24)end;
```

Xét tiêu chuẩn slicing *(n, {v}) = (19, {z}).* *n = 19* là vị trí của câu lệnh, *v = z* là biến của câu lệnh *n*

```
(1) begin
(2)   c = 1 ;
(3)   b = (2 * a) ;
(4)   x = 3 ;
(5)   z = 90 ;
(6)   if (b > c ) then
(7)     begin
(8)       z = (z + x) ;
(9)       while ( z > 30) do
(10)        begin
(11)          x = (y + 1) ;
(12)          y = (((y - z) + 3) * y) ;
(13)          z = (z - 3) ;
(14)        end;
(15)      od;
(16)    end;
(17)  else
(18)    begin
(19)      z = (z - x) ;
(20)    end;
(21)  fi;
(22)  tmpvar = x;
(23)  emptyline=1;
(24)end;
```


StaticSlicer 1.01
File
Help
Code:
(23,{z})
begin
c = 1 ;
b = (2 * a) ;
x = 3 ;
z = 90 ;
if (b > c) then
begin
z = (z + x) ;
while (z > 30) do
begin
x = (y + 1) ;
y = (((y - z) + 3) * y) ;
z = (z - 3) ;
end;
od;
end;
else
begin
z = (z - x) ;
end;
fi;
tmpvar = x;
emptyline=1;
end;
Lines in Slice:
[2,3,4,5,6,8,9,13,19]
Calculate slice!
Clear!

4.2. Chương trình Kaveri

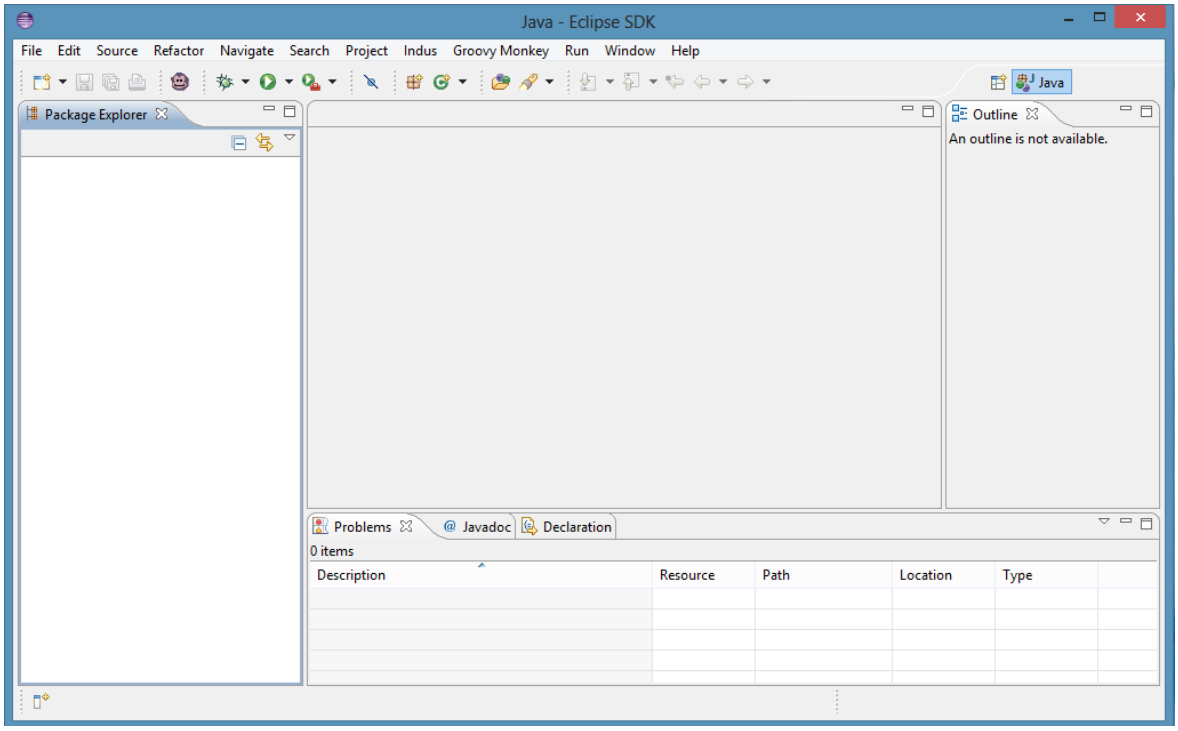
Yêu cầu để chạy chương trình *Kaveri*:

Eclipse: Eclipse phiên bản 3.0.x.

Indus Plug-in: Indus Plug-in phiên bản 0.6.x

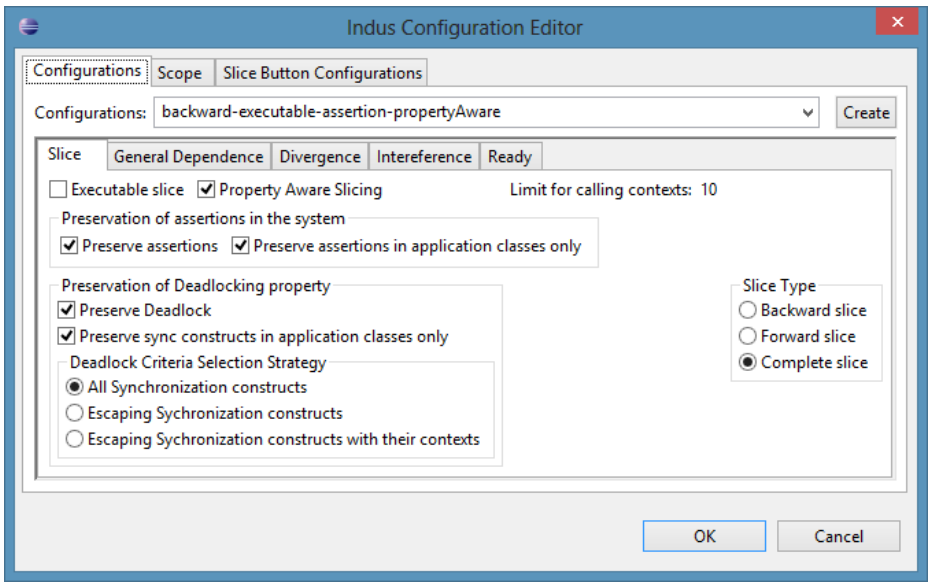
GroovyMonkey phiên bản 0.6.x

Khi cài xong chương trình sẽ có giao diện như sau:



Bài toàn như sau: Chương trình tính Tổng giai thừa các số từ 1 đến n.

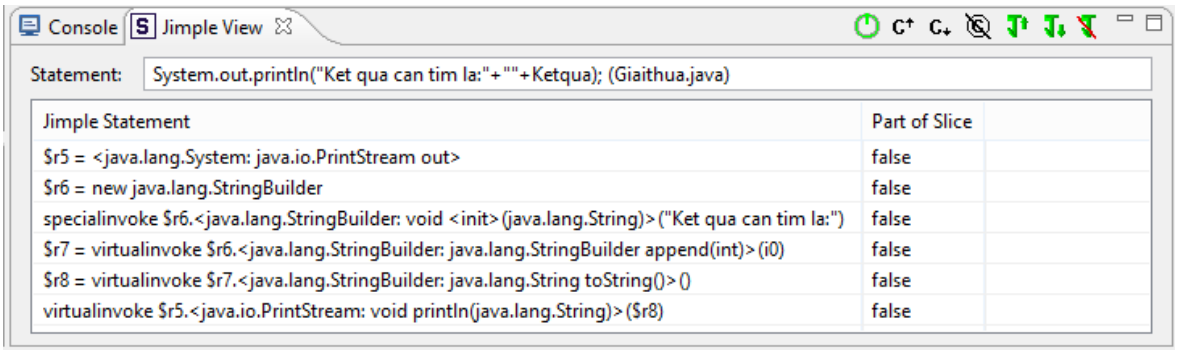
Để cấu hình các *slice slcing* ta vào *Indus Configuration* trên thanh *menu*.



Ta có code java của bài toán:

```
import java.io.*;
public class Giaithua {
public static int n,Sn;
	public Giaithua() {
		// TODO Auto-generated constructor stub
	}

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		int i=0;
		int j=0;
		int Ketqua=0;
		System.out.print("Nhap so n: ");
		try{
		n = System.in.read();
		}
		catch(IOException ex)
		{
			System.out.println("Khong doc duoc thong tin");

		}

		for(i=1;i<=n;i++)
		{
			int Sn=1;
			for(j=1;j<=i;j++)
			{
				Sn=Sn*j;
			}
			Ketqua=Sn+Ketqua;
		}
		System.out.println("Ket qua can tim la:"+""+Ketqua);

	}

}
```

Sau khi có bài toán ta cần thêm các tiêu chuẩn *slicing*. Để thêm các tiêu chuẩn *slicing* ta vào **Window -> Show View -> Other -> Kaveri ->Jimple View.** Khi đó ta sẽ có bảng dùng để thêm vào các tiêu chuẩn *slicing.*



Ta dùng các nút trong bảng *Jumple Statement* để thêm vào các tiêu chuẩn.Thứ tự các nút từ trái qua phải là:

Track Java Statements: Câu lệnh được chọn chỉ hiện thị khi bật hoặc tắt nút này.

Add as criteria (Pre-Execution): Thêm các tiêu chuẩn của câu lệnh liên quan tới chương trình cần slicing. Tiêu chuẩn được thêm vào dùng để điều khiển câu lệnh khi được bật nút.

Add as criteria (Post-Execution): Thêm các tiêu chuẩn của câu lệnh liên quan tới chương trình cần slicing. Tiêu chuẩn bổ sung thêm giá trị của biểu thức nếu đượcbật nút.

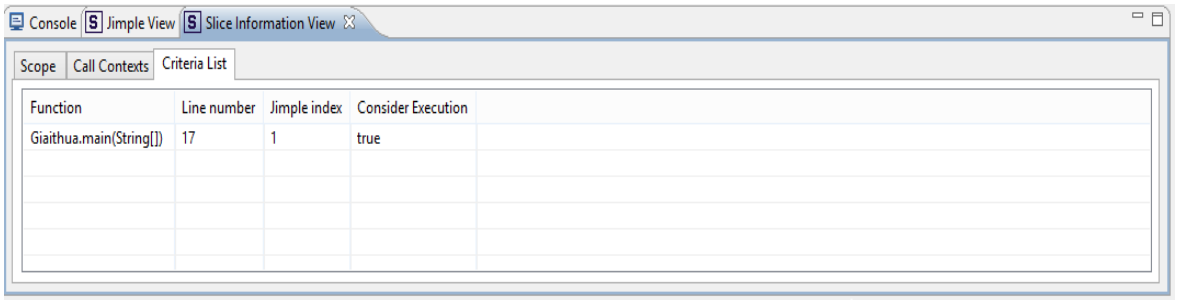
Remove Criteria: Loại bỏ các câu lệnh có liên quan đến tiêu chuẩn của bài toán.

Add as Java criteria (Pre-Execution): Nút này tương tự như nút *Add as criteria (Pre-Execution)* nhưng dùng để thêm tất cả các tiêu chuẩn một lần bật nút.

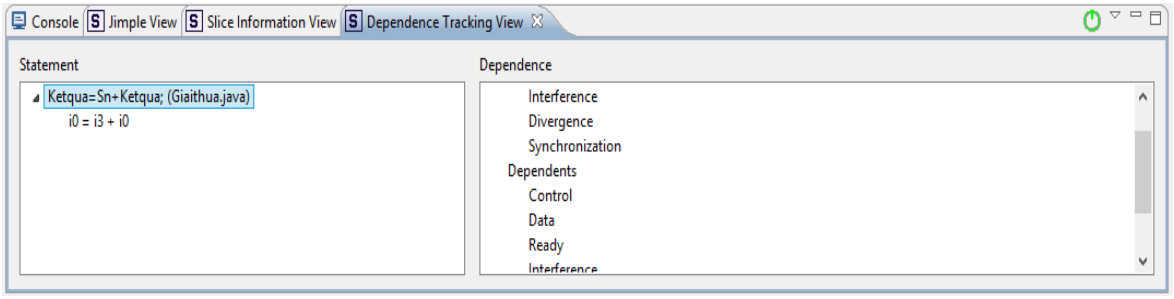
Add as Java criteria (Post-Execution): Nút này tương tự như nút *Add as criteria (Post-Execution)* nhưng dùng để thêm tất cả các tiêu chuẩn một lần bật nút.

Remove all Criteria: Loại bỏ tất cả các câu lệnh liên quan tới tiêu chuẩn của bài toán.

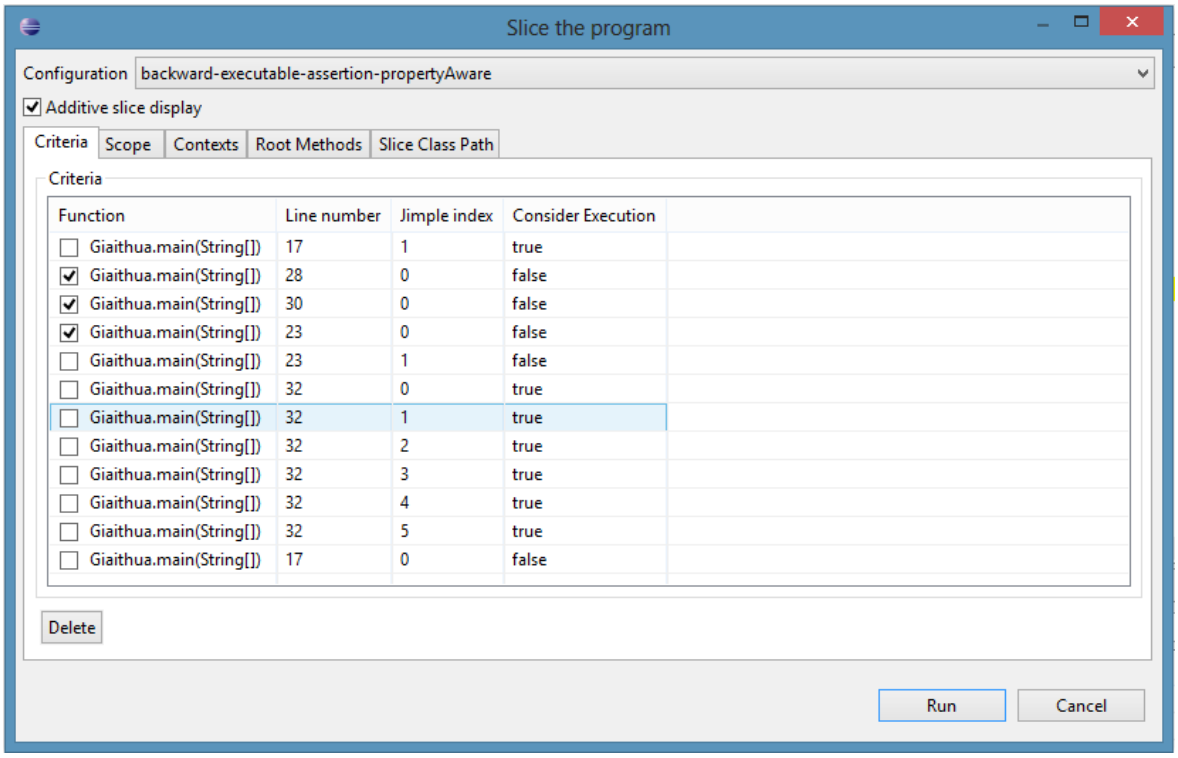
Để xem các tiêu chuẩn ta vào **Window -> Show View -> Other -> Kaveri ->Slice information View**.



Để xem các phụ thuộc của tiêu chuẩn được chọn ta vào **Window -> Show View -> Other -> Kaveri ->Dependence Tracking View**.



Chạy Kaveri: Kích chuột phải vào Java file hoặc Java project trong *navigator* hoặc *package explorer* trong giao diện của *Eclipse* và chọn **Indus -> Slice Java file or Indus -> Slice project.** Tại đây ta có thể chọn các tiêu chuẩn để thực hiện slicing từ các bước trên.Sau khi chọn xong các tiêu chuẩn ta kích vào *run* để chạy chương trình slicing cho bài toán.



KẾT LUẬN

Với một yêu cầu lớn vậy em chỉ có thể quan tâm được các vấn đề liên quan tới lập trình, lệnh trong chương trình, coding và kiểm thử, kiểm chứng.

Để thực hiện được các vấn đề trên đòi hỏi rất nhiều thời gian, phải theo dõi chặt chẽ các biến hay các câu lệnh. Đối với một chương trình nhỏ thì vấn đề này có thể dễ dàng thực hiện được nhưng với những chương trình lớn có hàng nghìn câu lệnh, nhiều biến chạy suốt dọc chương trình thì việc theo dõi một biến trở nên khó khăn không thể làm được hoặc làm được nhưng tốn rất nhiều thời gian.

Vì vậy đã có rất nhiều các kĩ thuật ra đời nhằm giải quyết vấn đề trên.Trong đó có kĩ thuật *Program Slicing*

Program slicing là một kĩ thuật lấy từ chương trình ra chả các câu lệnh ảnh hưởng đến một giá trị tính toán cụ thể tại các điểm được quan tâm gọi là các tiêu chuẩn slicing. *Program slicing* đã và đang được triển khai và áp dụng trong nhiều lĩnh vực công nghệ phần mềm như hỗ trợ gỡ rối chương trình kiểm thử, bảo trì phần mềm….

Hiện nay có nhiều phương pháp slicing khác nhau nhưng thường dùng kỹ thuật *static slicing* và *dynamic slicing* trên những chương trình có cấu trúc. Các kỹ thuật slicing thông dụng hiện nay để sử dụng luồng dữ liệu và đồ thị phụ thuộc chương trình để tính toán ra các slice slicing. Ngoài ra trong một số trường hợp cụ thể ta có thể sử dụng các kỹ thuật *program slicing* khác như slicing dựa trên luồng thông tin, dựa trên quan hệ phụ thuộc.

TÀI LIỆU THAM KHẢO

[Wei1] M. Weiser. Program slicing.

[Wei2] M. Weiser Program slices: formal, psychological, and practical investigations of an automatic program abstraction method . PhD thesis, University of Michigan, Ann Arbor, 1979.

[3] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. In Proceedings of the ACM SIGSOFT/SIGPLANSoftware Engineering Symposium on Practical Software Development Environments.

[4] J.-F. Bergeretti and B.A. Carre. Information- ow and data- ow analysis of while programs. ACM Transactions on Programming Languages and Systems.

[5] B. Korel and J. Laski. Dynamic slicing of computer programs. Journal of Systems and Software.

[6] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In Proceedings of the14th International Conference on Software Engineering.

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems.

[8] H. Agrawal and J.R. Horgan. Dynamic program slicing. In Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation.

[9] H. Agrawal, R.A. DeMillo, and E.H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In Proceedings of the ACM Fourth Symposium on Testing, Analysis, and Verification (TAV4).

[10] Frank Tip, A Survey of Program Slicing Techniques,