LỜI MỞ ĐẦU

Trong ngành kỹ nghệ phần mềm, năm 1979, có một quy tắc nổi tiếng là: "Trong một dự án lập trình điển hình, thì xấp xỉ 50% thời gian và hơn 50% tổng chi phí được sử dụng trong kiểm thử các chương trình hay hệ thống đã được phát triển". Và cho đến nay, sau gần một phần 3 thế kỷ, quy tắc đó vẫn còn đúng. Đã có rất nhiều ngôn ngữ, hệ thống phát triển mới với các công cụ tích hợp cho các lập trình viên sử dụng phát triển ngày càng linh động. Nhưng kiểm thử vẫn đóng vai trò hết sức quan trọng trong bất kỳ dự án phát triển phần mềm nào.

Rất nhiều các giáo sư, giảng viên đã từng than phiền rằng: "Sinh viên của chúng ta tốt nghiệp và đi làm mà không có được những kiến thực thực tế cần thiết về cách để kiểm thử một chương trình. Hơn nữa, chúng ta hiếm khi có được những lời khuyên bổ ích để cung cấp trong các khóa học mở đầu về cách một sinh viên nên làm về kiểm thử và gỡ lỗi các bài tập của họ".

Các tác giả của cuốn sách nổi tiếng "The Art of Software Testing" – Nghệ thuật kiểm thử phần mềm, Glenford J. Myers, Tom Badgett, Todd M. Thomas, Corey Sandler đã khẳng định trong cuốn sách của mình rằng: " Hầu hết các thành phần quan trọng trong các thủ thuật của một nhà kiểm thử chương trình là kiến thức về cách để viết các ca kiểm thử có hiệu quả". Việc xây dựng các testcase là một nhiệm vụ rất khó khăn. Để có thể xây dựng được tập các testcase hữu ích cho kiểm thử, chúng ta cần rất nhiều kiến thức và kinh nghiệm.

Đó là những lý do thúc đẩy chúng em thực hiện đề tài này. Mục đích của đề tài là tìm hiểu những kiến thức tổng quan nhất về kiểm thử, và cách thiết kế testcase trong kiểm thử phần mềm. Việc thực hiện đề tài sẽ giúp em tìm hiểu sâu hơn và lĩnh vực rất hấp dẫn này, vận dụng được các kiến thức đã học để có thể thiết kế được các testcase một cách có hiệu quả và áp dụng vào những bài toán thực tế.

LỜI CẢM ƠN

Trước hết, chúng em xin chân thành cảm ơn thầy giáo, TS. Nguyễn Văn Hiệp đã chỉ bảo và giúp đỡ tận tình cho chúng em trong giai đoạn luận văn tốt nghiệp này; cũng như sự góp ý của thầy giáo phản biện, PGS. TS Quản Thành Thơ và sự giúp đỡ của mọi người đã giúp chúng em hoàn thiện đề tài luận văn này.

Chúng em hi vọng sẽ nhận được sự đóng góp ý kiến của các thầy cô và các bạn để bản báo cáo được hoàn thiện hơn. Những đóng góp đó sẽ là kinh nghiệm quý báu cho chúng em. Và từ đó, chúng em có thể tiếp tục phát triển đề tài này.

Chúng em xin chân thành cảm ơn.

TP. HỒ CHÍ MINH, 1/ 2014

Sinh viên:

Cao Trọng Đại

Phạm Công Cương

GIỚI THIỆU

Bản báo cáo này trình bày quá trình xây dựng chương trình tự động sinh testcase cho một hàm chức năng theo ngôn ngữ Java, sử dụng kỹ thuật kiểm thử hộp trắng theo dòng điều khiển. Phạm vi hoạt động của chương trình:

- Tạo đồ thị dòng điều khiển cơ bản và tập đường thi hành tuyến tính độc lập cho hầu hết các hàm, nhưng chỉ tạo được testcase cho các hàm chứa tham số đầu vào kiểu Int.
- Hàm được chọn không được chứa các hàm được gọi từ bên ngoài.
- Xử lý được các cấu trúc điều khiển if-else, for, while, do-while, switch, break, continue.
- Các biểu thức quan hệ trong biểu thức điều kiện được sử lý bao gồm NOT, AND, OR.

Báo cáo luận văn được trình bày theo cấu trúc gồm 4 chương. Trong đó:

- Chương 1 sẽ giới thiệu sơ qua về ngành kiểm thử phần mềm, sự thiết yếu và các phương pháp kiểm thử.
- Chương 2 sẽ trình bày các kiến thức về kỹ thuật kiểm thử luồng điều khiển, từ đó hình thành ý tưởng để xây dựng chương trình sinh testcase.
- Chương 3 trình bày hiện thực chương trình dựa vào kiến thức ở chương 2.
- Chương 4 nói về kết quả thực hiện được và hướng phát triển của chương trình.

Mục Lục:

Danh mục hình ảnh:

CHƯƠNG 1: TỔNG QUAN VỀ KIỂM THỬ PHẦN MỀM

1.1 Sản phẩm phần mềm:

Phần mềm là một (bộ) chương trình được cài đặt trên máy tính nhằm thực hiện một nhiệm vụ tương đối độc lập nhằm phục vụ cho một ứng dụng cụ thể việc quản lý hoạt động của máy tính hoặc áp dụng máy tính trong các hoạt động kinh tế, quốc phòng, văn hóa, giáo dục, giải trí,…

Lỗi phần mềm: là sự không khớp giữa chương trình và đặc tả của nó.

Dựa vào định nghĩa, chúng ta có thể thấy lỗi phần mềm xuất hiện theo ba dạng sau:

- Sai: Sản phẩm được xây dựng khác với đặc tả.
- Thiếu: Một yêu cầu đã được đặc tả nhưng lại không có trong sản phẩm được xây dựng.
- Thừa: Một yêu cầu được đưa vào sản phẩm mà không có trong đặc tả. Cũng có trường hợp yêu cầu này có thể là một thuộc tính sẽ được người dùng chấp nhận nhưng khác với đặc tả nên vẫn coi là có lỗi.
- Một hình thức khác nữa cũng được xem là lỗi, đó là phần mềm khó hiểu, khó sử dụng, chậm hoặc dễ gây cảm nhận rằng phần mềm hoạt động không đúng.

1.2 Kiểm thử phần mềm là gì[1]:

Kiểm thử phần mềm là qui trình chứng minh phần mềm không có lỗi.

Mục đích của kiểm thử phần mềm là chỉ ra rằng phần mềm thực hiện đúng các chức năng mong muốn.

Kiểm thử phần mềm là qui trình thiết lập sự tin tưởng về việc phần mềm hay hệ thống thực hiện được điều mà nó hỗ trợ.

Kiểm thử phần mềm là qui trình thi hành phần mềm với ý định tìm kiếm các lỗi của nó.

Kiểm thử phần mềm được xem là qui trình cố gắng tìm kiếm các lỗi của phần mềm theo tinh thần "hủy diệt".

Các mục tiêu chính của kiểm thử phần mềm:

- Phát hiện càng nhiều lỗi càng tốt trong thời gian kiểm thử xác định trước.
- Chứng minh rằng sản phẩm phần mềm phù hợp với các đặc tả yêu cầu của nó.
- Xác thực chất lượng kiểm thử phần mềm đã dùng chi phí và nỗ lực tối thiểu.
- Tạo các testcase chất lượng cao, thực hiện kiểm thử hiệu quả và tạo ra các báo cáo vấn đề đúng và hữu dụng.

Kiểm thử phần mềm là 1 thành phần trong lĩnh vực rộng hơn, đó là Verification & Validation (V &V), ta tạm dịch là Thanh kiểm tra và Kiểm định phần mềm.

- Thanh kiểm tra phần mềm là qui trình xác định xem sản phẩm của 1 công đoạn trong qui trình phát triển phần mềm có thoả mãn các yêu cầu đặt ra trong công đoạn trước không (Ta có đang xây dựng sản phẩm một cách đúng đắn không ?) Các hoạt động Thanh kiểm tra phần mềm bao gồm kiểm thử (testing) và xem lại (reviews).
- Kiểm định phần mềm là qui trình đánh giá phần mềm ở cuối chu kỳ phát triển để đảm bảo sự hài lòng của khách hàng khi sử dụng sản phẩm. (Ta có xây dựng phần mềm đúng theo yêu cầu khách hàng?).

1.3 Tại sao phải kiểm thử phần mềm:

Mọi hành động của con người đều có thể gây ra lỗi, tất cả các chương trình, máy móc, thiết bị, vật dụng,...do con người làm ra đều có thể có lỗi, rõ ràng hoặc tiềm ẩn. Vì vậy hoạt động kiểm thử phần mềm là việc cần nên làm. Công việc kiểm thử phần mềm có thể được thực hiện ở bất kỳ tại các khâu sản xuất phần mềm như lấy yêu cầu khách hàng, phân tích yêu cầu, thiết kế, coding và triển khai phần mềm đến người dùng. Tùy vào giai đoạn kiểm thử phần mềm sẽ có các phương pháp kiểm thử khác nhau.

1.4 Các mức độ kiểm thử phần mềm[1]:

Kiểm thử đơn vị (Unit Testing): kiểm thử sự hiện thực chi tiết của từng đơn vị nhỏ (hàm, class,...) có hoạt động đúng không?

Kiểm thử module (Module Testing): kiểm thử các dịch vụ của module có phù hợp với đặc tả của module đó không?

Kiểm thử tích hợp (Integration Testing): kiểm thử xem từng phân hệ của phần mềm có đảm bảo với đặc tả thiết kế của phân hệ đó không?

Kiểm thử hệ thống (System Testing): kiểm thử các yêu cầu không chức năng của phần mềm như hiệu suất, bảo mật, làm việc trong môi trường căng thẳng,...

Kiểm thử độ chấp nhận của người dùng (Acceptance Testing): kiểm tra xem người dùng có chấp thuận sử dụng phần mềm không?

Kiểm thử hồi qui: được làm mỗi khi có sự hiệu chỉnh, nâng cấp phần mềm với mục đích xem phần mềm mới có đảm bảo thực hiện đúng các chức năng trước khi hiệu chỉnh không?

1.5 Testcase và các phương pháp thiết kế testcase[1]:

Testcase: Mỗi testcase chứa các thông tin cần thiết để kiểm thử thành phần phần mềm theo 1 mục tiêu xác định.

Phương pháp thiết kế testcase: Bất kỳ sản phẩm kỹ thuật nào (phần mềm không phải là ngoại lệ) đều có thể được kiểm thử bởi 1 trong 2 cách:

- Kiểm thử hộp đen (Black box testing): theo góc nhìn sử dụng. Không cần kiến thức về chi tiết thiết kế và hiện thực bên trong. Kiểm thử dựa trên các yêu cầu và đặc tả sử dụng TPPM.
- Kiểm thử hộp trắng (White box testing) : theo góc nhìn hiện thực. Cần kiến thức về chi tiết thiết kế và hiện thực bên trong. Kiểm thử dựa vào phủ các lệnh, phủ các nhánh, phủ các điều kiện con,..

Ứng với mỗi mức độ kiểm thử mà ta áp dụng các phương pháp kiểm thử khác nhau, được mô tả trong hình 1.1:

Kiểu kiểm thử	Kỹ thuật kiểm thử được dùng
Unit Testing	White Box, Black Box
Integration Testing	Black Box, White Box
Functional Testing	Black Box
System Testing	Black Box
Accceptance Testing	Black Box

Hình 1.1 Phương pháp kiểm thử cho từng mức độ kiểm thử

CHƯƠNG 2: KỸ THUẬT KIỂM THỬ LUỒNG ĐIỀU KHIỂN

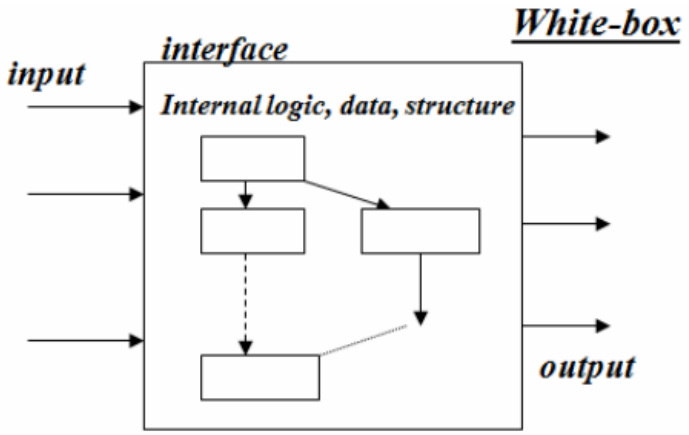
2.1 Kỹ thuật kiểm thử hộp trắng[1]:

Đối tượng được kiểm thử là 1thành phần phần mềm (TPPM). TPPM có thể là 1 hàm chức năng, 1 module chức năng, 1 phân hệ chức năng…

Kiểm thử hộp trắng dựa vào thuật giải cụ thể, vào cấu trúc dữ liệu bên trong của đơn vị phần mềm cần kiểm thử để xác định đơn vị phần mềm đó có thực hiện đúng không.

Do đó người kiểm thử hộp trắng phải có kỹ năng, kiến thức nhất định về ngôn ngữ lập trình được dùng, về thuật giải được dùng trong TPPM để có thể thông hiểu chi tiết về đoạn code cần kiểm thử.

Thường tốn rất nhiều thời gian và công sức nếu thành phần phần mềm quá lớn. Do đó kỹ thuật này chủ yếu được dùng để kiểm thử đơn vị.



Hình 2.2 Kiểm thử hộp trắng.

Có 2 hoạt động kiểm thử hộp trắng :

- Kiểm thử luồng điều khiển : tập trung kiểm thử thuật giải chức năng.
- Kiểm thử dòng dữ liệu : tập trung kiểm thử đời sống của từng biến dữ liệu được dùng trong thuật giải.

2.2 Đường thi hành (Execution path)[1]:

Là 1 kịch bản thi hành đơn vị phần mềm tương ứng, cụ thể nó là danh sách có thứ tự các lệnh được thi hành ứng với 1 lần chạy cụ thể của đơn vị phần mềm, bắt đầu từ điểm nhập của đơn vị phần mềm đến điểm kết thúc của đơn vị phần mềm.

Mỗi TPPM có từ 1 đến n (có thể rất lớn) đường thi hành khác nhau. Mục tiêu của phương pháp kiểm thử luồng điều khiển là đảm bảo mọi đường thi hành của đơn vị phần mềm cần kiểm thử đều chạy đúng. Rất tiếc trong thực tế, rất khó để thực hiện điều này ngay cả trên những đơn vị phần mềm nhỏ.

Như đoạn code sau chỉ có 1 đường thi hành nhưng 1 tỷ lệnh gọi doSomething:

```
for (i=1; i<=1000; i++)
for (j=1; j<=1000; j++)
for (k=1; k<=1000; k++)
doSomethingWith(i,j,k);
```

Còn đoạn code gồm 32 lệnh if else độc lập sau :

```
if (c1) s11 else s12;
if (c2) s21 else s22;
if (c3) s31 else s32;
...
if (c32) s321 else s322;
```

Có 2^32 = 4 tỉ đường thi hành khác nhau.

Mà cho dù có kiểm thử hết được toàn bộ các đường thi hành thì vẫn không thể phát hiện những đường thi hành cần có nhưng không (chưa) được hiện thực :

```
if (a>0) doIsGreater();
if (a==0) dolsEqual();
// thiếu việc xử lý trường hợp a < 0 - if (a<0) dolsLess();
```

Một đường thi hành đã kiểm tra là đúng nhưng vẫn có thể bị lỗi khi dùng thật (trong 1 vài trường hợp đặc biệt) :

```
int phanso (int a, int b) {
return a/b;
}
```

Khi kiểm tra, ta chọn b <> 0 thì chạy đúng, nhưng khi dùng thật trong trường hợp b = 0 thì hàm phanso bị lỗi.

2.3 Các cấp phủ kiểm thử (Coverage) [1]:

Trong thực tế, để giảm bớt công sức và thời gian nhưng vẫn đảm bảo kết quả, ta nên kiểm thử tập testcase tối thiểu mà đạt kết quả tối đa. Để đo độ tin cậy của tập testcase, người ta đưa ra khái niệm phủ kiểm thử.

Phủ kiểm thử: là tỉ lệ các thành phần thực sự được kiểm thử so với tổng thể sau khi đã kiểm thử các test case được chọn. Phủ càng lớn thì độ tin cậy càng cao.

Phủ cấp 0: kiểm thử những gì có thể kiểm thử được, phần còn lại để người dùng phát hiện và báo lại sau. Đây là mức độ kiểm thử không thực sự có trách nhiệm.

Phủ cấp 1: kiểm thử sao cho mỗi lệnh được thực thi ít nhất 1 lần.

Phân tích hàm foo sau đây :

```
1     float foo(int a, int b, int c, int d){
2     float e;
3     if(a==0)
4     return 0;
5     int x = 0;
6     if((a==b) || ((c==d) && bug(a)))
7     x = 1;
8     e = 1/x;
9     return e;
10    }
```

Với hàm foo trên, ta chỉ cần 2 test case là đạt 100% phủ cấp 1:

- foo(0,0,0,0) trả về 0

- foo(1,1,1,1) trả về 1

Phủ cấp 2: kiểm thử sao cho mỗi điểm quyết định luận lý đều được thực hiện ít nhất 1 lần cho trường hợp TRUE lẫn FALSE. Ta gọi mức kiểm thử này là phủ các nhánh (Branch coverage). Phủ các nhánh đảm bảo phủ các lệnh.

Line	Predicate	TRUE	FALSE
3	a==0	TC 1: foo(0, 0, 0, 0) return 0	TC 2: foo(1, 1, 1, 1) return 1
6	(a==b) OR ((c == d) AND bug(a))	TC 2: foo(1, 1, 1, 1) return 1	TC 3: foo(1, 2, 1, 2) division by zero!

Bảng 2.1 Testcase phủ cấp 2

Với test case 1 và test case 2 ta chỉ đạt phủ 2/3 phủ cấp 2. Nếu thêm test case 3 thì mới đạt 100% phủ cấp 2.

Phủ cấp 3: kiểm thử sao cho mỗi điều kiện luận lý con (subcondition) của từng điểm quyết định đều được thực hiện ít nhất 1 lần cho trường hợp TRUE lẫn FALSE. Ta gọi mức kiểm thử này là phủ các điều kiện con. Phủ các điều kiện con chưa chắc đảm bảo phủ các nhánh và ngược lại.

Predicate	TRUE	FALSE
a==0	TC 1 : foo(0, 0, 0, 0) return 0	TC 2 : foo(1, 1, 1, 1) return 1
a==b	TC 2 : foo(1, 1, 1, 1) return 1	TC 3 : foo(1, 2, 1, 2) division by zero!
c==d	TC 4 : foo(1, 2, 1, 1) Return 1	TC 3 : foo(1, 2, 1, 2) division by zero!
bug(a)	TC 4 : foo(1, 2, 1, 1) Return 1	TC 5 : foo(2,1, 1, 1) division by zero!

Bảng 2.2 Test case phủ cấp 3

Phủ cấp 4: kiểm thử sao cho mỗ điều kiện luận lý con (subcondition) của từng điểm quyết định đều được thực hiện ít nhất 1 lần cho trường hợp TRUE lẫn FALSE và điểm quyết định cũng được kiểm thử cho cả 2 nhánh TRUE lẫn FALSE. Ta gọi mức kiểm thử này là phủ các nhánh và các điều kiện con (branch & subcondition coverage). Đây là mức độ phủ kiểm thử tốt nhất trong thực tế.

2.4 Đồ thị dòng điều khiển[1]:

2.4.1 Khái niệm đồ thị dòng điều khiển:

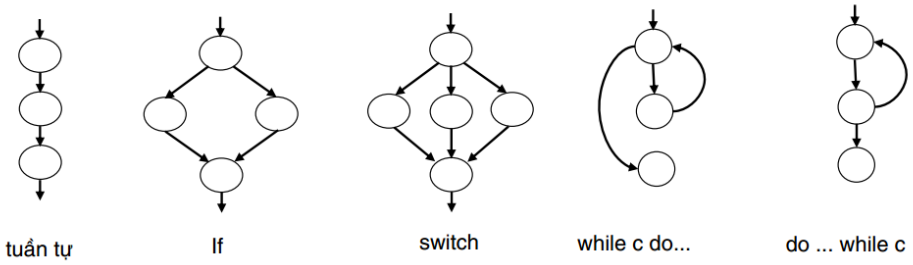
Là một trong nhiều phương pháp miêu tả thuật giải. Đây là phương pháp trực quan cho chúng ta thấy dễ dàng các thành phần của thuật giải và mối quan hệ trong việc thực hiện các thành phần này. Các thành phần này có thể là một câu lệnh đơn giản, cũng có thể là một khối lệnh trong mã nguồn.

Gồm 2 loại thành phần: các nút và các cung nối giữa chúng

Các loại nút trong đồ thị dòng điều khiển:

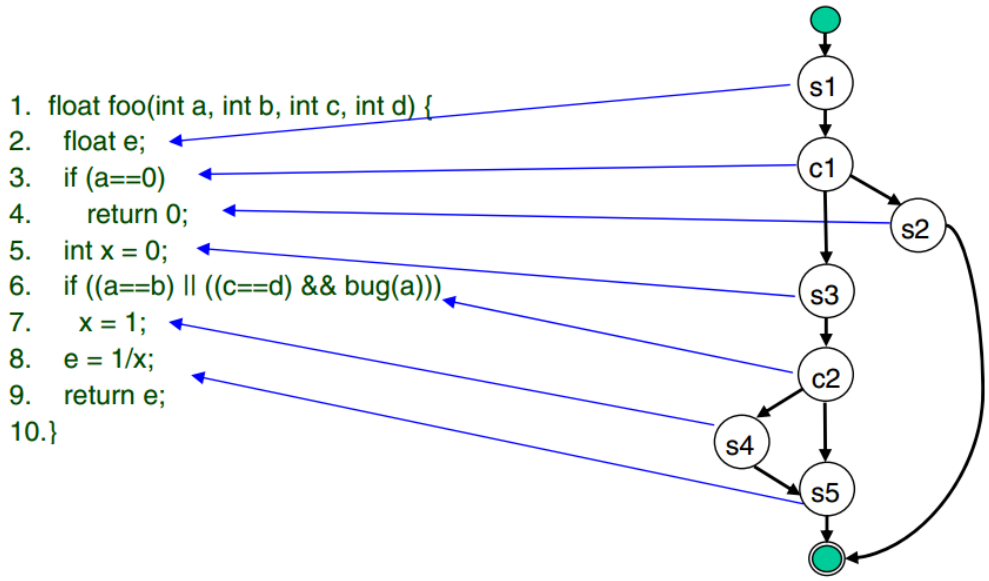
Hình 2.3 Các loại nút trong đồ thị dòng điều khiển.

Từ các nút trên, ta sẽ xây dựng sơ đồ dòng điều khiển cho mã nguồn qua việc xử lý các cấu trúc điều khiển. Các cấu trúc điều khiển phổ dụng được miêu tả bằng sơ đồ dòng điều khiển như sau:



Hình 2.4 Cấu trúc điều khiển.

Ví dụ:



Hình 2.5 Ví dụ về đồ thị dòng điều khiển.

Như vậy, việc thể hiện mã nguồn của một thành phần phần mềm dưới dạng đồ thị dòng điều khiển, ta sẽ dễ dàng tìm đường được thi hành của mã nguồn đó. Như ví dụ trong hình 2.4, nhìn vào mã nguồn ta sẽ rất khó xác định đường thi hành nhưng khi nhìn vào đồ thị ta dễ dàng hơn trong việc xác định đường thi hành của đoạn mã nguồn.

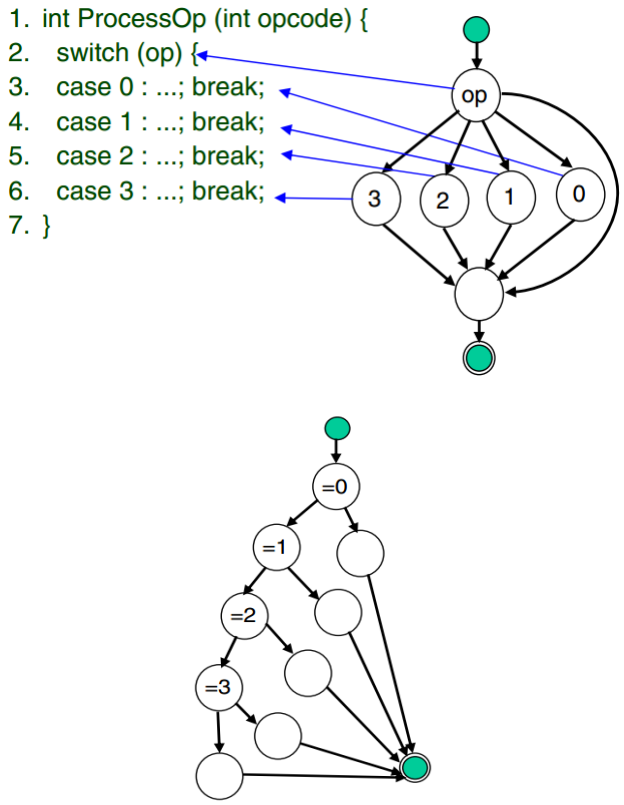
2.4.2 Đồ thị dòng điều khiển nhị phân:

Tiêu chuẩn để phân biệt đồ thị dòng điều khiển đó là dựa vào tính chất các nút của đồ thị, mà cụ thể các nút quyết định. Nếu đồ thị có chứa bất kỳ một nút quyết định đa phân nào đó thì gọi là đồ thị dòng điều khiển đa phân. Nếu đồ thị chỉ chứa các nút quyết định nhị phân thì gọi là đồ thị dòng điều khiển nhị phân.

Ngoài ra đồ thị còn được đánh giá dựa vào tính chất của các đường thi hành. Chẳng hạn như một đồ thị chỉ chứa các đường thi hành tuyến tính độc lập.

Nhằm đơn giản hóa đồ thị dòng điều khiển và giúp cho quá trình tìm kiếm đường thi hành đơn giản hơn, người ta thường đưa về đồ thị dòng điều khiển đơn giản hơn. Chẳng hạn như từ đa phân về nhị phân.

Tuy nhiên trong mã nguồn hầu như chỉ chứa một nút đa phân là câu lệnh switch, do đó khi đưa từ đồ thị dòng điều khiển đa phân về nhị phân ta chỉ cần xử lý khối đa phân của câu lệnh switch như sau:



Hình 2.5 Chuyển câu lệnh switch từ đa phân sang nhị phân.

Như hình 2.5, sau khi xử lý câu lệnh switch ta được một đồ thị chỉ chứa nút quyết định nhị phân.

2.4.3 Đồ thị dòng điều khiển cơ bản:

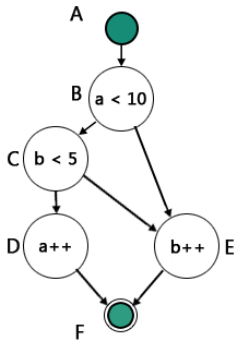
Trong đồ thị dòng điều khiển nhị phân, nếu từng nút quyết định chỉ chứa một điều kiện con luận lý thì ta nói đồ thị này là đồ thị dòng điều khiển cơ bản. Trong đồ thị này chỉ chưa các đường thi hành cơ bản.

Ta luôn có thể chi tiết hóa một đồ thị dòng điều khiển bất kỳ thành đồ thị dòng điều khiển nhị phân. Tương tự, ta luôn có thể chi tiết hóa một đồ thị dòng điều khiển nhị phân bất kỳ thành một đồ thị dòng điều khiển cơ bản. Dựa vào đường thi hành cơ bản ta sẽ luôn tìm được tập các đường thi hành tốt nhất để xây dựng testcase.

Ví dụ đồ thị dòng điều khiển cơ bản cho một đoạn mã nguồn đơn giản:

```
If( a < 10 && b < 5)
      a++;
else
```

```
b++;
```



Hình 2.6 Ví dụ đồ thị dòng điều khiển cơ bản.

Tóm lại, ta luôn có thể chi tiết hóa một đồ thị dòng điều khiển bất kỳ thành đồ thị dòng điều khiển cơ bản.

2.4.4 Đường thi hành tuyến tính độc lập:

Một đồ thị dòng điều khiển sẽ giúp ta nhận biết được các đường thi hành, mục đích của ta là xử lý các đường thi hành này và sinh testcase. Như vậy muốn xây dựng một tập testcase tối ưu nhất, có nghĩa là số testcase nhỏ nhất đạt độ bao phủ tốt nhất thì phải xây dựng được tập đường thi hành tối ưu.

Trong thực tế, tập các đường thi hành tối ưu chính là tập các đường thi hành tuyến tính độc lập cơ bản.

Một đường thi hành gọi là độc lập khi nó có ít nhất một lệnh mới hoặc điều kiện mới so với các đường thi hành khác trong tập các đường thi hành. Đối với đồ thị dòng điều khiển, một đường thi hành bao gồm các nút và cạnh, do đó một đường thi hành độc lập sẽ phải có ít nhất một cạnh mới không nằm trong các đường thi hành còn lại. Nếu là đồ thị dòng điều khiển cơ bản, tập các đường thi hành độc lập này sẽ trở thành tập các đường thi hành độc lập cơ bản.

Một đường thi hành được gọi là tuyến tính độc lập khi nó là một đường thi hành không thể sinh ra từ phép toán tuyến tính giữa các đường khác trong tập. Thực ra, đây là

một cách định nghĩa theo toán học của đường thi hành độc lập. Ví dụ trong hình 2.6, ta xây dựng tập các đường thi hành tuyến tính độc lập gồm 3 đường sau:

- Đường 1: (A, B, C, D, F)
- Đường 2: (A, B, C, E, F)
- Đường 3:(A, B, E, F)

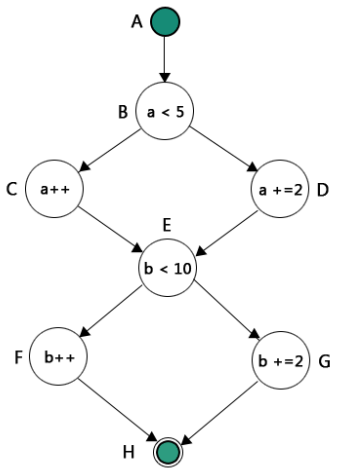
Nếu một tập các đường thi hành tuyến tính độc lập được xây dựng dựa trên đồ thị dòng điều khiển cơ bản, thì tập này gọi là tuyến tính độc lập cơ bản.

Trong ví dụ hình 2.6 thì chỉ có một tập đường thi hành tuyến tính độc lập cơ bản duy nhất. Tuy nhiên đối với một đồ thị dòng điều khiển cơ bản có thể có nhiều hơn một tập đường thi hành tuyến tính độc lập.

Ví dụ:

```
If(a < 5) a++; else a+=2;
If(b < 10) b++; else b+=2;
```

Đồ thị dòng điều khiển ứng với đoạn mã trên như sau:



Hình 2.7 Ví dụ đồ thị dòng điều khiển cơ bản.

Với đồ thị dòng điều khiển hình 2.7 ta sẽ xây dựng nhiều tập đường thi hành tuyến tính độc lập cơ bản khác nhau, tuy nhiên các tập này đều tối ưu như nhau, tức số đường thi hành ít nhất nhưng đạt mức độ bảo phủ tốt nhất – phủ cấp 4.

Tập 1 gồm các đường thi hành:

- Đường 1: (A, B, C, E, F, H)
- Đường 2: (A, B, C, E, G, H)
- Đường 3: (A, B, D, E, F, H)

Tập 2 gồm các đường thi hành:

- Đường 1: (A, B, C, E, F, H)
- Đường 2: (A, B, D, E, F, H)
- Đường 3: (A, B, D, E, G, H)

Nếu ví dụ trên ta chọn thêm một đường thi hành nữa để cho vào tập 1 chẳng hạn như tập (A, B, D, E, G, H), thì tập này sẽ không còn là tuyến tính độc lập cơ bản nữa. Bởi vì đường thi hành (A, B, D, E, G, H) không chứa một lệnh nào mới so với ba đường còn lại. Về mặt toán học thì đường này có thể được sinh ra từ phép toán tuyến tính giữa ba đường còn lại.

Khi xác định tập các đường thi hành tuyến tính độc lập cơ bản từ đồ thị dòng điều khiển cơ bản ta sẽ gặp phải một số khó khăn. Thứ nhất là xác định được số đường thi hành trong tập, thứ hai là trong các đường này phải đảm bảo là tuyến tính độc lập cơ bản.

2.4.5 Độ phức tạp Cyclomatic:

Để thuận tiện hơn trong việc xác định số đường thi hành tối thiểu trong tập đường thi hành tuyến tính độc lập cơ bản mà vẫn đảm bảo phủ cấp 4, Tom McCabe đã đưa ra một định nghĩa gọi là độ phức tạp Cyclomatic, ký hiệu là C = V(G) của đồ thị dòng điều khiển, nhằm xác định được số đường thi hành tuyến tính độc lập.

Tom McCabe đã đưa ra các công thức sau để tính V(G) cho một đồ thị dòng điều khiển:

- V(G) = E – N + 2, E là số cung, N là số nút của đồ thị.
- V(G) = P + 1, Nếu là đồ thị dòng điều khiển nhị phân và P nút quyết định

Trong ví dụ 2.6, dựa vào công thức ta dễ dàng xác định được C = 3 với hai nút quyết định nhị phân B, C. Với ví dụ 2.7, C = 3 với hai nút quyết định B, E. Như vậy khi xây dựng được đồ thị dòng điều khiển ta cũng dễ dàng xác định được cần bao nhiêu testcase

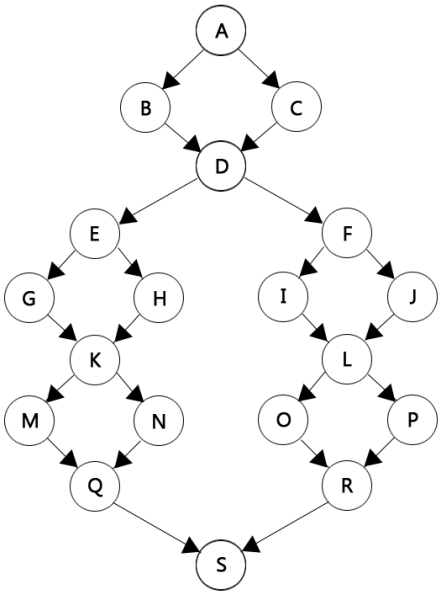
để tối ưu trong công việc kiểm thử. Đối với đồ thị dòng điều khiển cơ bản, số C sẽ là số các đường thi hành tuyến tính độc lập cơ bản, nếu lựa chọn được đúng C đường tuyến tính độc lập và kiểm thử tất cả các đường này, ta sẽ đạt được phủ cấp 4, là cấp thử tốt nhất trong thực tế.

2.4.6 Quy trình xác định C đường tuyến tính độc lập:

Để xác định C đường tuyến tính độc lập, Tom McCabe đề nghị quy trình gồm các bước sau:

1. Xác định đường tuyến tính đầu tiên bằng cách đi dọc theo nhánh bên trái nhất của các nút quyết định. Chọn đường này là pilot.
2. Dựa trên đường pilot, thay đổi cung xuất của nút quyết định đầu tiên và cố gắng giữ lại maximum phần còn lại.
3. Dựa trên đường pilot, thay đổi cung xuất của nút quyết định thứ 2 và cố gắng giữ lại maximum phần còn lại.
4. Tiếp tục thay đổi cung xuất cho từng nút quyết định trên đường pilot để xác định đường thứ 4, 5,... cho đến khi không còn nút quyết định nào trong đường pilot nữa.
5. Lặp chọn tuần tự từng đường tìm được làm pilot để xác định các đường mới xung quanh nó y như các bước 2, 3, 4 cho đến khi không tìm được đường tuyến tính độc lập nào nữa (khi đủ số C).

Ví dụ sử dụng quy trình này để xác định tập các đường tuyến tính độc lập cho đồ thị dòng điều khiển bên dưới.



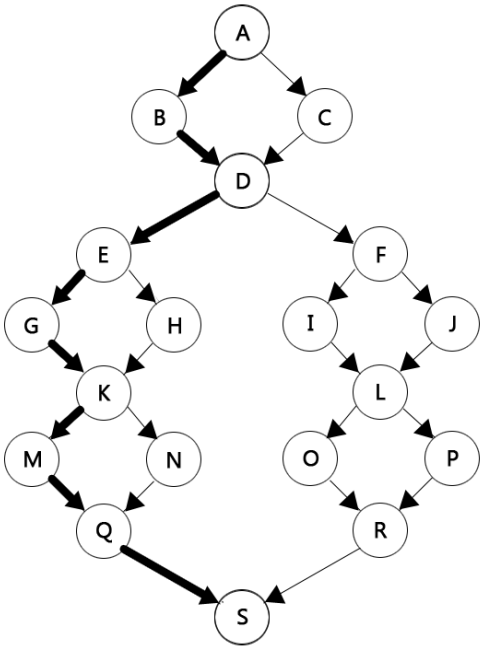
Hình 2.8 Ví dụ xác định đường độc lập truyến tính.

Với đồ thị dòng điều khiển hình 2.8, độ phức tạp Cyclomatic được tính như sau:

C = V(G) = E – N + 2 = 24 – 19 – 2 = 7

C = V(G) = P + 1 = 6 + 1 = 7

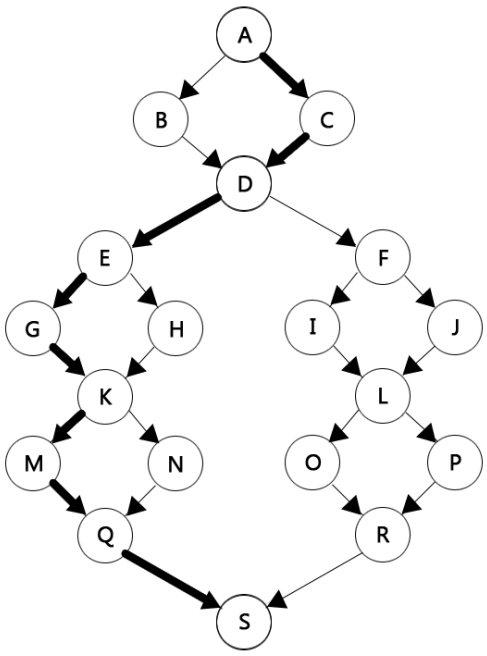
Sau đó ta xây dựng tập gồm 7 đường thi hành tuyến tính độc lập. Theo quy trình Tom McCabe, đầu tiên ta chọn đường thi hành bên trái nhất làm đường pilot.



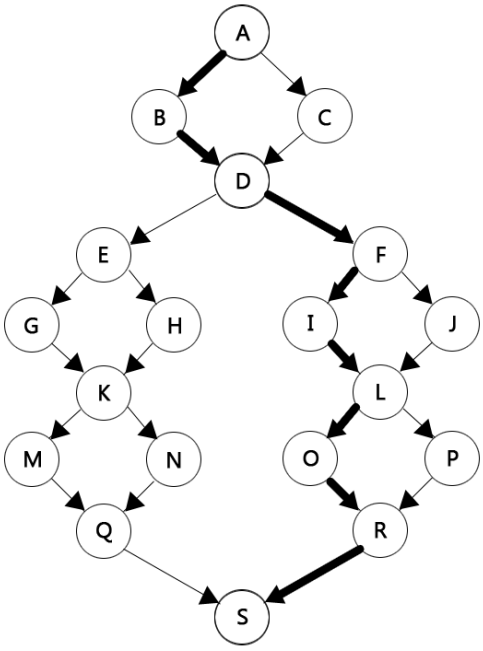
Hình 2.9 Đường pilot đầu tiên.

Để xác định đường thi hành tuyết tính độc lập thứ 2 ta thay đổi nút quyết định đầu tiên của đường pilot là nút A và giữ tối ta phần còn lại (Hình 2.10).

Để xác định đường thi hành tuyến tính độc lập thứ 3, ta sẽ thay đổi nút quyết định thứ 2 của đường pilot là nút D, thay đổi nút này không thể giữ nguyên phần còn lại (Hình 2.11).



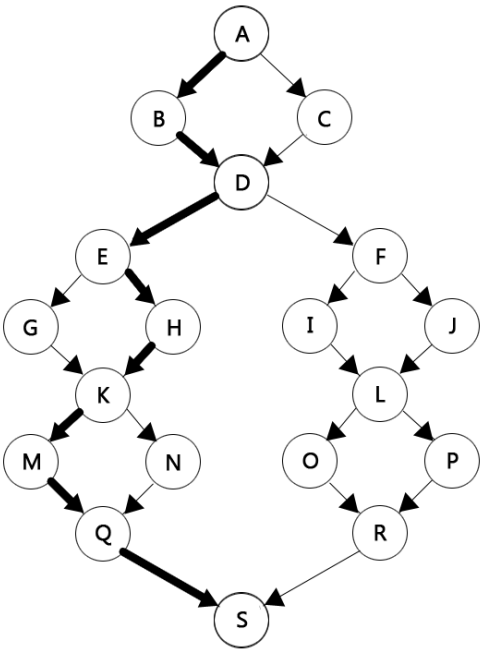
Hình 2.10 Đường tuyến tính độc lập thứ 2.



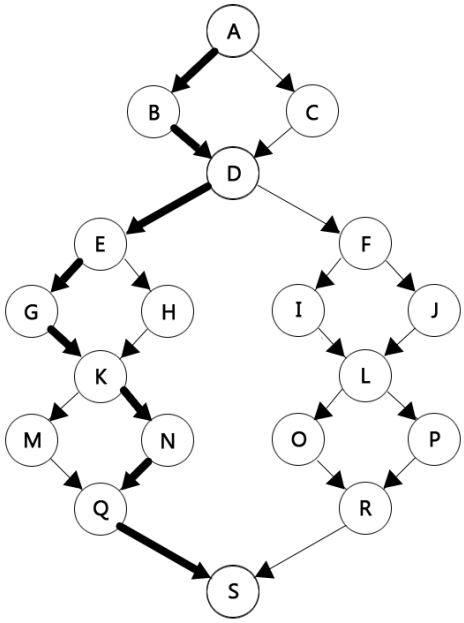
Hình 2.11 Đường tuyến tính độc lập thứ 3.

Tương tự đường tuyến tính độc lập thứ 4 ta thay đổi nút E của đường pilot (Hình 2.12).

Đường thứ 5 ta thay đổi nút K của đường pilot (Hình 2.13).

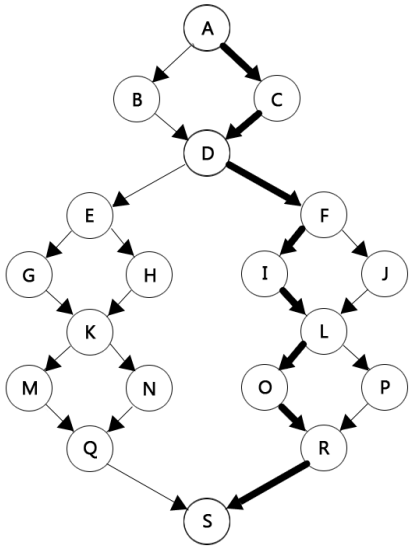


Hình 2.12 Đường tuyến tính độc lập thứ 4.

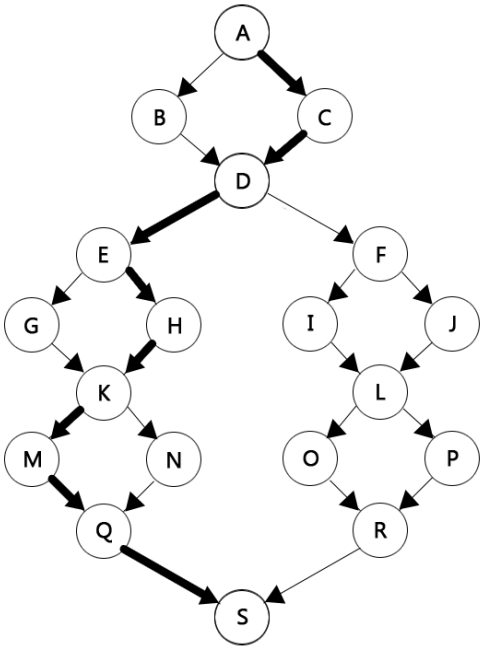


Hình 2.13 Đường tuyến tính độc lập thứ 5.

Sau khi xử lý nút K, thì đường pilot đầu tiên sẽ không còn nút quyết định nào nữa, ta sẽ chọn một đường khác làm pilot và thay đổi các nút quyết định trên đường pilot mới để tìm ra các đường thi hành tuyến tính độc lập mới. Quá trình sẽ kết thúc khi ta tìm đủ số lượng đường thi hành tuyến tính độc lập đúng bằng số C. Trong ví dụ này ta sẽ tìm thêm hai đường tuyến tính độc lập dựa trên đường thứ 2 làm đường pilot để hoàn thành tập các đường thi hành tuyến tính độc lập (Hình 2.14, Hình 2.15).



Hình 2.14 Đường tuyến tính độc lập thứ 6.



Hình 2.15 Đường tuyến tính độc lập thứ 7.

Như vậy, sau khi kết thúc quá trình ta có được tập các đường thi hành tuyến tính độc lập như sau:

- Đường 1: (A, B, D, E, G, K, M, Q, S)
- Đường 2: (A, C, D, E, G, K, M, Q, S)
- Đường 3: (A, B, D, F, I, L, O, R, S)
- Đường 4: (A, B, D, E, H, K, M, Q, S)
- Đường 5: (A, B, D, E, G, K, N, Q, S)
- Đường 6: (A, C, D, F, I, L, O, R, S)
- Đường 7: (A, C, D, E, H, K, M, Q, S)

Đối với ví dụ này, sẽ có những tập các đường thi hành tuyến tính độc lập khác nhau, tuy nhiên các tập này đều có độ tối ưu như nhau, nghĩa là đảm bảo số đường thi hành ít nhất nhưng phủ tốt nhất.

2.5 Kiểm thử hộp trắng[1]:

2.5.1 Quy trình kiểm thử hộp trắng:

Phần này sẽ trình bày quy trình kiểm thử hộp trắng dựa vào dòng điều khiển được Tom McCabe đề xuất vào giữa thập niên 1970. Hay còn biết đến như kiểm thử cấu trúc.

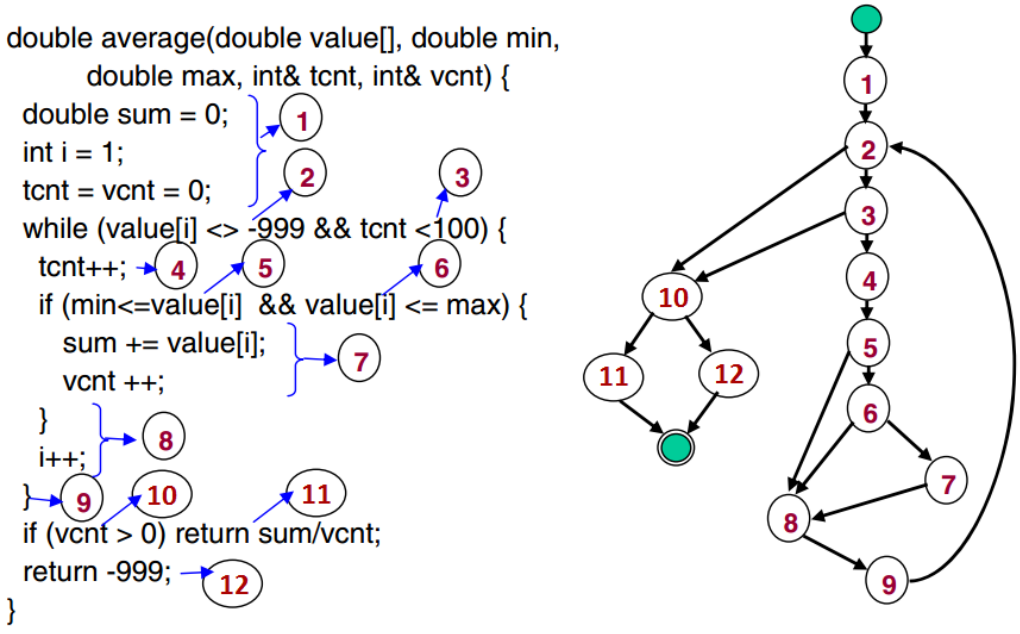
Quy trình kiểm thử gồm các bước sau:

1. Từ TPPM cần kiểm thử, xây dựng đồ thị dòng điều khiển tương ứng, rồi chuyển thành đồ thị dòng điều khiển nhị phân, rồi chuyển thành đồ thị dòng điều khiển cơ bản.
2. Tính độ phức tạp Cyclomatic của đồ thị C = P +1(Mục 2.6).
3. Xác định C đường thi hành tuyến tính độc lập cơ bản cần kiểm thử (Mục 2.4.6).
4. Tạo từng test case cho từng đường thi hành tuyến tính độc lập cơ bản.
5. Thực hiện kiểm thử trên từng test case.
6. So sánh kết quả có được với kết quả được kỳ vọng.
7. Lập báo cáo kết quả để phản hồi cho những người có liên quan.

Trong phạm vi của bài báo cáo này, ta sẽ hiện thực tới bước 4, là tạo testcase cho từng đường thi hành tuyến tính độc lập, ở đây, testcase là những điều kiện đầu vào của mã nguồn để đảm bảo đường thi hành tuyến tính độc lập này sẽ được thực hiện khi kiểm thử thành phần phần mềm.

2.5.2 Ví dụ về kỹ thuật kiểm thử dòng điều khiển:

Ví dụ sau đây sẽ minh họa cụ thể kỹ thuật kiểm thử dòng điều khiển một đơn vị phần mềm nhỏ.



Hình 2.16 Ví dụ kiểm thử dòng điều khiển.

Đồ thị trên có 5 nút quyết định nhị phân nên có độ phức tạp C=5+1=6.

6 đường thi hành tuyến tính độc lập cơ bản là :

1. 1→2→10→11
2. 1→2→3→10→11
3. 1→2→3→4→5→8→9
4. 1→2→3→4→5→6→8→9
5. 1→2→3→4→5→6→7→8→9
6. 1→2→10→12

Phân tích mã nguồn của hàm average, ta định nghĩa 6 testcase kết hợp với 6 đường thi hành tuyến tính độc lập cơ bản như sau :

Test case cho đường 1 :

- value(k) <>-999, với 1< k < i
- value(i) = -999 với 2 ≤ i ≤ 100
- Kết quả kỳ vọng : (1) average = Giá trị trung bình của i-1 giá trị hợp lệ. (2) tcnt = i-1. (3) vcnt = i-1
- Chú ý : không thể kiểm thử đường 1 này riêng biệt mà phải kiểm thử chung với đường 4 hay 5 hay 6.

Test case cho đường 2 :

- value(k) <>-999, với ∀k < i , i >100
- Kết quả kỳ vọng : (1) average = Giá trị trung bình của 100 giá trị hợp lệ. (2) tcnt = 100. (3) vcnt = 100

Test case cho đường 3:

- value(i) <> -999 ∀ i <= 100 và value(k) < min với k < i
- Kết quả kỳ vọng : (1) average = Giá trị trung bình của n giá trị hợp lệ. (2) tcnt = 100. (3) vcnt = n (số lượng giá trị hợp lệ)

Test case cho đường 4:

- value(i) <>-999 với ∀ i <= 100
- và value(k) > max với k <= i
- Kết quả kỳ vọng : (1) average = Giá trị trung bình của n giá trị hợp lệ. (2) tcnt = 100. (3) vcnt = n (số lượng giá trị hợp lệ)

Test case cho đường 5:

- value(i) <>-999 và min <= value(i) <= max với ∀ i <= 100
- Kết quả kỳ vọng : (1) average = Giá trị trung bình của 100 giá trị hợp lệ. (2) tcnt = 100. (3) vcnt = 100

Test case cho đường 6:

- value(1) = -999
- Kết quả kỳ vọng : (1) average = -999. (2) tcnt = 0 (3) vcnt = 0

Như vậy chúng ta vừa tìm hiểu cơ sở lý thuyết về kỹ thuật kiểm thử dòng điều khiển. Theo phương pháp này, chúng ta sẽ xây dựng đồ thị dòng điều khiển của thành phần phần mềm. Sau đó chuyển đồ thị này về dạng nhị phân, sau đó từ dạng nhị phân ta chuyển về đồ thị dòng điều khiển cơ bản. Sau đó sinh testcase cho C đường độc lập tuyến tính cơ bản của đồ thị dòng điều khiển cơ bản.

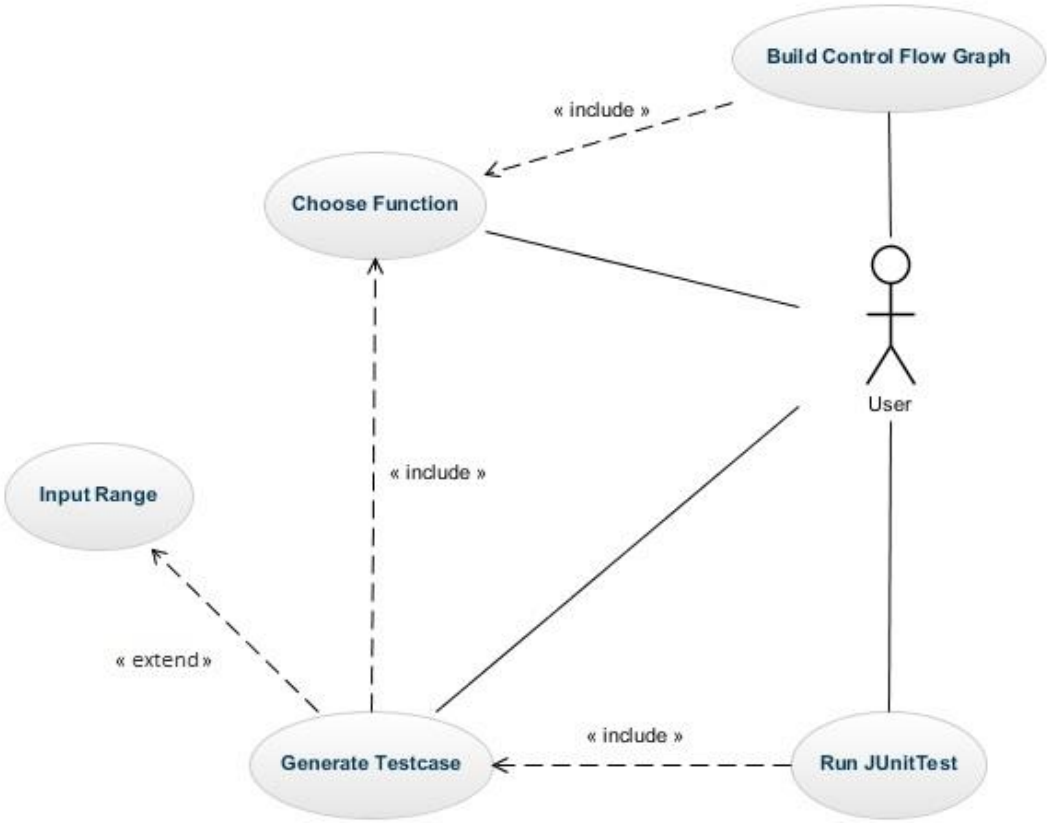
CHƯƠNG 3: XÂY DỰNG ĐỒ THỊ DÒNG ĐIỀU KHIỂN CƠ BẢN VÀ SINH TESTCASE

Chương này sẽ trình bày phương pháp hiện thực một chương trình nhỏ theo hướng kiểm thử hộp trắng, cụ thể là nguyên lý kiểm thử dòng điều khiển đã được trình bày tại mục 2.5.1

3.1 Sơ đồ tổng quát hiện thực chương trình:

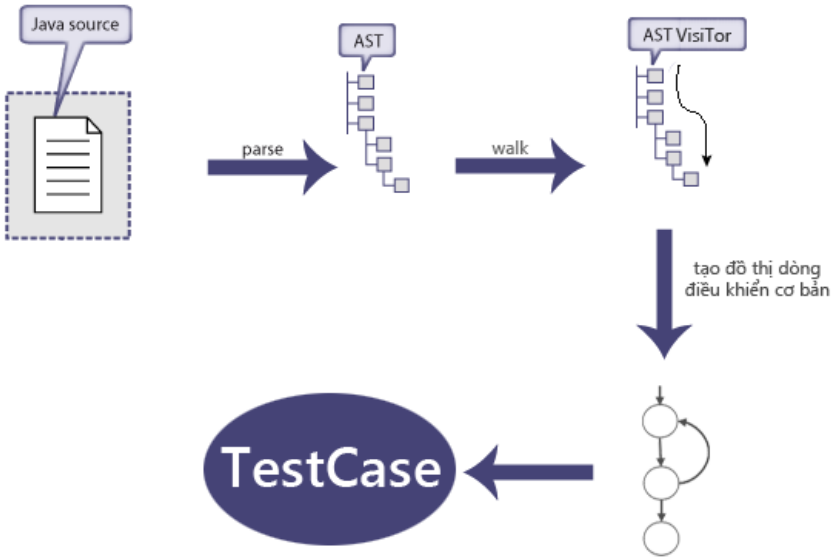
Nhóm sẽ xây dựng chương trình dưới dạng một plug-in cho Eclipse IDE. Chương trình này cho phép chọn một hàm tại một Java source code và chạy sinh testcase. Testcase đầu ra dưới dạng là một file đầu vào của JUnit test, do đó ta có thể thực hiện kiểm tra testcase ngay lập tức.

Sơ đồ use-case của chương trình như sau:



Hình 3.1 Sơ đồ use-case.

Sơ đồ tổng quát hiện thực chương trình như sau:



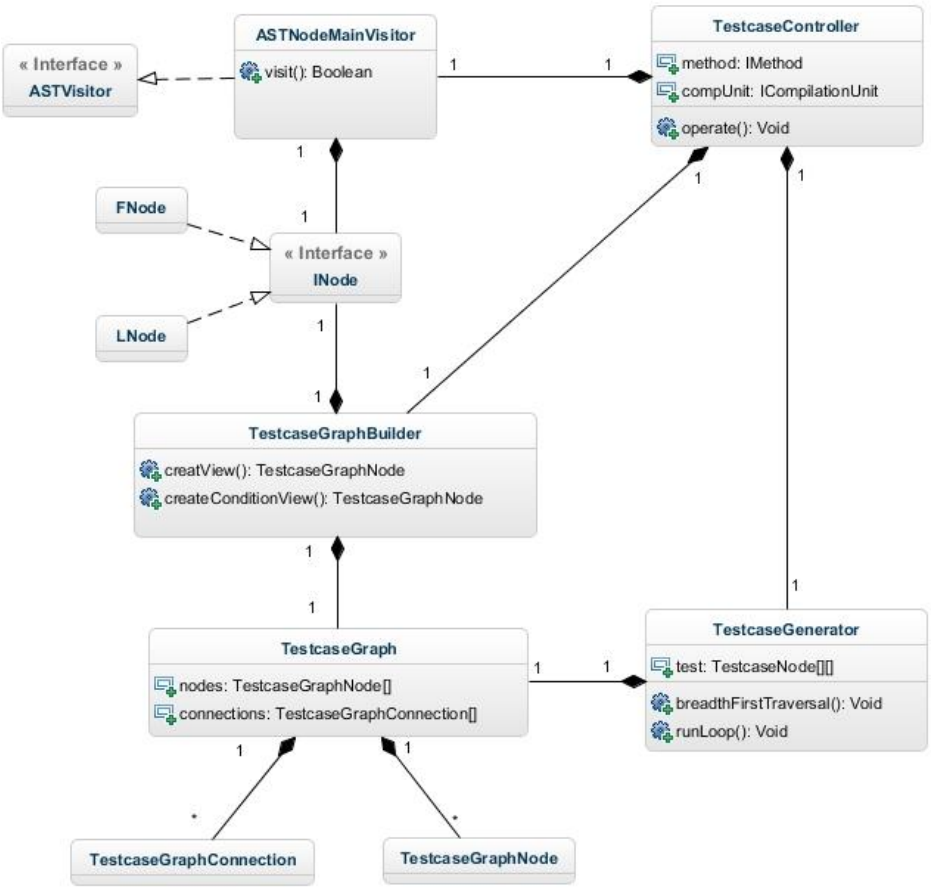
Hình 3.2 Sơ đồ hiện thực chương trình.

Theo sơ đồ ta sẽ tiến hành xây dựng chương trình tạo testcase theo 3 bước:

- Bước 1: Ta tiến hành phân tích mã nguồn của thành phần phần mềm thành cây cú pháp trừu tượng (Abstract Syntax Tree – AST).
- Bước 2: Duyệt cây cú pháp thu được và xử lý các nút của AST, từ đó tạo nền tảng cho việc tạo đồ thị dòng điều khiển cơ bản.
- Bước 3: Từ đồ thị dòng điều khiển cơ bản tìm các đường thi hành tuyến tính độc lập cơ bản và sinh TestCase.

Chương trình sẽ được xây dựng theo cấu trúc được thể hiện ở sơ đồ class sau. Qua sơ đồ này ta nắm bắt được những class quan trọng sẽ được thực thi trong chương trình.

Sơ đồ quan hệ class:



Hình 3.3 Sơ đồ quan hệ lớp.

3.2 Tạo cây cú pháp trừu tượng:

3.2.1 Cây cú pháp trừu tượng – AST:

Cây cú pháp trừu tượng là dạng mã nguồn đã được phân tích từ vựng và ngữ pháp để chuyển thành dạng có cấu trúc nhằm xử lý mã nguồn dưới dạng gần với mã máy hơn.

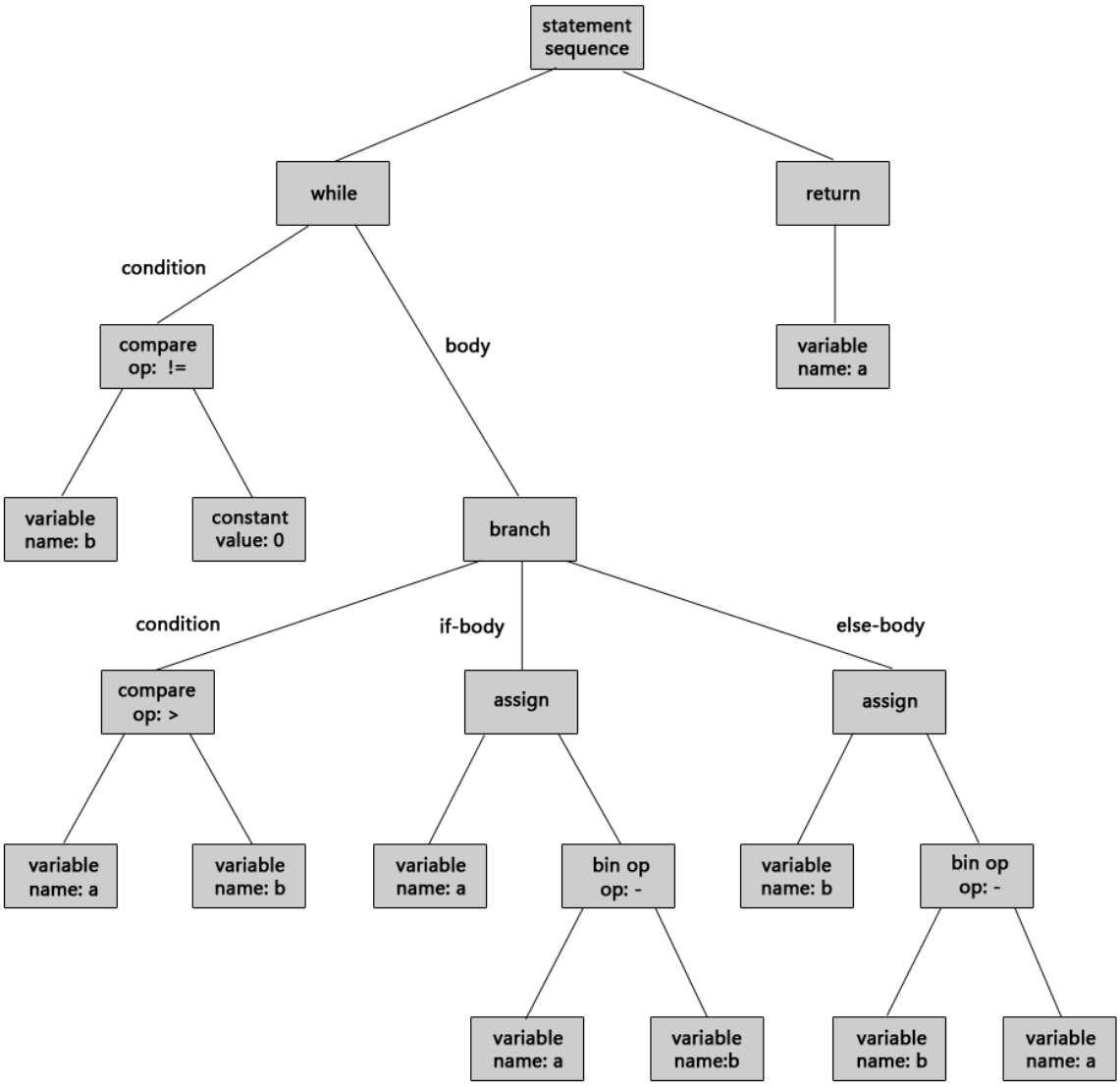
Đối với chương trình tạo testcase, cây cú pháp trừu tượng là nền tảng để giúp xử lý mã nguồn nhằm xây dựng đồ thị dòng điều khiển cơ bản cho mã nguồn.

Sau đây là một ví dụ về cây cú pháp[2]:

```
while( b != 0 ){
      if( a > b ) a := a – b;
      else b := b – a;
```

```
}
```

Sau khi phân tích từ vựng và ngữ pháp ta được cây cú pháp như sau:



Hình 3.4: Ví dụ về cây cú pháp.

3.2.2 Tạo cây cú pháp từ mã nguồn:

Để tạo cây cú pháp cho một ngôn ngữ chúng ta phải xây dựng được bộ phân tích từ vựng và cú pháp cho ngôn ngữ đó. Từ bộ phân tích này chúng ta sẽ xây dựng được cây cú pháp.

Nhóm đã tìm hiểu hai giải pháp để xây dựng được cây cú pháp từ mã nguồn:

- Sử dụng AST của Eclipse Java Development Tools (JDT)[9]: công cụ này cung cấp các API để truy xuất và thao tác mã nguồn Java.
    - ✓ Ưu điểm: Cung cấp sẵn khả năng phân tích ngôn ngữ Java một cách nhanh chóng và chính xác.
    - ✓ Nhược điểm: Chỉ sử dụng được cho ngôn ngữ Java.
- Sử dụng công cụ ANTLR - Another Tool For Language Recognition[8]: là một công cụ sinh ra bộ phân tích từ vựng và ngữ pháp của một ngôn ngữ một cách từ động từ bộ luật sinh của ngôn ngữ cần phân tích.
    - ✓ Ưu điểm: Có thể xây dựng được bộ phân tích cú pháp cho nhiều ngôn ngữ khác nhau. Hỗ trợ một số công cụ cho phép thể hiện giao diện trực quan.
    - ✓ Nhược điểm: So với AST thì phải xây dựng được được bộ phân tích cho ngôn ngữ cần phân tích.

Trong phạm vi luận văn, chương trình sẽ tạo testcase cho ngôn ngữ Java. Vì vậy, với khả năng cung cấp sẵn bộ phân tích dành cho ngôn ngữ Java của AST, nhóm quyết định chọn giải pháp sử dụng AST của Eclipse Java Development Tools. Giải pháp này sẽ giúp nhóm tiết kiệm được nhiều thời gian cho việc xây dựng bộ phân tích, không những thế còn đảm bảo chính xác cho quá trình tạo cây cú pháp. Các bước sau của quá trình tạo chương trình cũng được thực hiện dựa trên giải pháp này.

A. Công cụ AST của Eclipse Java Development Tools[9]:

AST định nghĩa các API cho phép chỉnh sửa, tạo, đọc và xóa mã nguồn. Gói chính của AST nằm trong org.eclipse.jdt.core.dom của org.eclipse.jdt.core plug-in.

Mỗi mã nguồn Java được thể hiện như một cây của các node AST, các node này là lớp con của lớp ASTNode. Mỗi node này là đặc trưng cho một phần tử của ngôn ngữ Java. Ví dụ như, có các node cho khai báo phương thức (MethodDeclaration), khai báo biến (VariableDeclarationFragment),…

Để tạo cây cú pháp trừu tượng từ công cụ AST, ta sẽ sử dụng ASTParser. Nó sẽ xử lý toàn bộ mã nguồn Java cũng như các thành phần của mã Java.

Ví dụ bên dưới sẽ trình bày phương thức parse(ICompilationUnit unit), dùng để phân tích mã nguồn được lưu trong file mà tham số unit chỉ tới.

```
protected CompilationUnit parse(ICompilationUnit unit) {
	ASTParser parser = ASTParser.newParser(AST.JLS3);
	parser.setKind(ASTParser.K_COMPILATION_UNIT);
	parser.setSource(unit);
	return (CompilationUnit) parser.createAST(null);
}
```

Với ASTParser.newParser(AST.JLS3) ta chỉ rõ sẽ phân tích mã theo chuẩn của Java Language Specification, phiên bản thứ 3.

parser.setKind(ASTParser.K_COMPILATION_UNIT) nói với parser rằng đầu vào là ICompilationUnit, một ICompilationUnit là một con trỏ chỉ tời một file Java. Parser hỗ trợ 4 loại đầu vào khác nhau:

- K_COMPILATION_UNIT: đầu vào là một con trỏ chỉ tới một file Java.
- K_EXPRESSION: đầu vào chứa một biểu thức Java.
- K_STATEMENTS: đầu vào chứa một câu lệnh Java.
- K_CLASS_BODY_DECLARATIONS: đầu vào chứa các phần tử của một class Java như khai báo phương thức, khai báo biến,...

Sau đó ta sẽ chỉ định mã nguồn mà parser sẽ phân tích qua câu lệnh parser.setSource(unit). Và kết quả trả về của hàm là cây cú pháp trừu tượng AST được tạo thông qua lệnh parser.createAST(null).

Trong phương thức parse trên có tham số đầu vào là unit kiểu ICompilationUnit, nó đại diện cho toàn bộ một file Java. Một ICompilationUnit có thể được lấy theo nhiều cách khác nhau. Ta sẽ trình bày cách được sử dụng trong chương trình. Chương trình sẽ được khởi chạy như một action, được định nghĩa trong file plugin.xml.

```
<extension
	point="org.eclipse.ui.popupMenus">
	<objectContribution
```

```
        objectClass="org.eclipse.jdt.core.IMethod"
        id="PluginATG.contribution1">
        <menu
        label="Generate Testcase"
        path="additions"
        id="PluginATG.menu1">
        <separator
        name="group1">
        </separator>
        </menu>
        <action
        class="de.htwg.flowchartgenerator.popup.actions.MainAction"
        enablesFor="1"
        icon="icons/plugin-icon16.png"
        id="PluginATG.newAction"
        label="Build"
        menubarPath="PluginATG.menu1/group1">
        </action>
</extension>
```

Ta thêm phần mở rộng objectContribution vào point "org.eclipse.ui.popupMenus", thiết lập objectClass là "org.eclipse.jdt.core.IMethod". Ta có thể lấy được ICompilationUnit thông qua ISelection, được truyền vào delegate của action, trong trường hợp này là class MainAction. Trong class MainAction sẽ hiện thực các phương thức của interface IObjectActionDelegate.

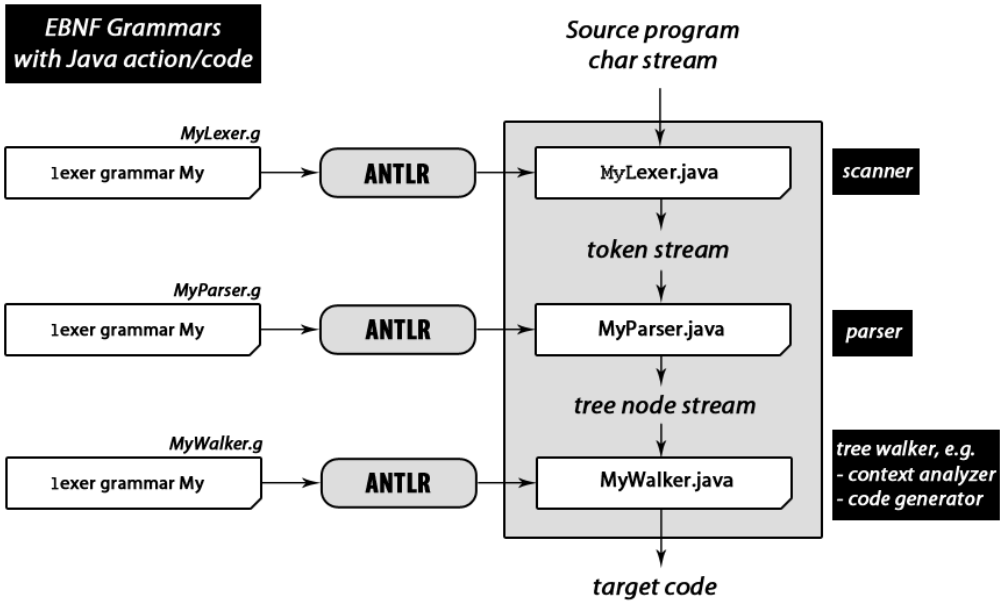
```
public void selectionChanged(IAction action, ISelection selection) {
        this.fSelection = selection;
}
private IMethod getCompilationUnit() {
        return (IMethod) ((IStructuredSelection) fSelection).getFirstElement();
}
public void run(IAction action) {
        ICompilationUnit cu = getCompilationUnit().getCompilationUnit();
}
```

Ta sẽ lấy được ISelection qua phương thức selectionChanged. Sau đó thông qua ISelection ta lấy được IMethod với phương thức getCompilationUnit(). Cuối cùng, từ IMethod ta có được ICompilationUnit với phương thức getCompilationUnit() của IMethod.

B. Công cụ ANTLR:

Là một công cụ sinh ra bộ phân tích từ vựng và ngữ pháp của một ngôn ngữ một cách từ động từ bộ luật sinh của ngôn ngữ cần phân tích. ANTLR được phát triển bởi Terance Parr từ nằm 1989 cùng các cộng sự của ông. Hiện tại công cụ này đã trở nên rất phổ biến với khoảng 5000 lượt tải về mỗi tháng.

Hình 3.3.1 cho ta thấy toàn bộ quá trình hoạt động của ANLTR trong việc sử lý một mã nguồn. Ta sẽ viết ba file là lexer, parser và tree grammar dưới dạng ngữ pháp EBNF, sau đó ba file này sẽ được antlr xử lý và sinh ra ba file java tương ứng là MyLexer.java, MyParser.java và MyWalker.java. Ba file này sẽ xử lý mã nguồn đầu vào và sinh ra bộ xử lý ngôn ngữ theo cách người lập trình định nghĩa.



Hình 3.5: Tổng quan về ANTLR.

Trong ANLTR, các file mô tả ngôn ngữ sẽ được viết theo ngữ pháp EBNF (Extended Backus-Naur Form) – là mở rộng của BNF với những phép toán mới. BNF – Backus-Naur Form là một cấu trúc các ký hiệu dùng để mô tả các ngữ pháp phi ngữ cảnh, do đó BNF thường được dùng để mô tả các ngôn ngữ lập trình. Ví dụ một luật sinh sẽ được mô tả như sau:

```
number : digit | number digit ;
digit : ‘0’|‘1’|‘2’|‘3’|‘4’|‘5’|‘6’|‘7’|‘8’|‘9’;
```

Với việc bổ sung những phép toán mới EBNF giúp cho việc mô tả các luật sinh trở nên ngắn gọn và dễ hiểu hơn BNF. Nhưng khả năng mô tả ngôn ngữ của EBNF là không hơn BNF, vì nếu luật sinh nào có thể mô tả được với EBNF thì cũng sẽ mô tả được với BNF. Các phép toán được thêm vào như ký hiệu “+”, “*”, “?”, [],... Ví dụ:

```
A+ : một hoặc nhiều A.
A* : không hoặc nhiều A.
A? : có hoặc không có A.
[‘0’..‘9’] : một trong các ký tự từ 0 đến 9.
```

Ví dụ muốn mô tả một phép tính cộng giữa nhiều số ta có thể viết một đoạn ngữ pháp EBNF như sau:

```
addOp : number( ‘+’ number)+;
number : digit+;
digit : [‘0’..‘9’];
```

Thông thường thì bộ phân tích từ vựng và ngữ pháp sẽ được viết riêng, sau khi mã nguồn được xử lý bởi lexer sẽ được chuyển thành dạng dòng từ tố, sau đó những từ tố này sẽ được parser xử lý. Tuy nhiên ANLTR hỗ trợ chúng ta sinh ra hai bộ phân tích này dựa trên các luật sinh của ngôn ngữ được mô tả trong một file duy nhất gọi là combined grammer. File này sẽ mô tả ngôn ngữ lập trình dưới dạng ngữ pháp EBNF, sau đó ANTLR sẽ tự động sinh bộ phân tích từ vựng là lexer và bộ phân tích ngữ pháp parser. Hai file này sẽ được tự động sinh ra theo một ngôn ngữ mà người dùng chọn trước như Java, C, C++, ... được gọi là ngôn ngữ đích.

Cấu trúc của một file combined grammar trong ANTLR sẽ có dạng như sau:

```
{gtype} grammar JavaMinus;
options{ // những thiết lập cho toàn bộ file, như thiết lập ngôn ngữ đích, hoặc thiết lập thông số cần cho quá trình sinh ra file lexer và parser từ combined grammar...}
tokens{// phần định nghĩa các từ tố}
@header{// chứa các header của java}
@rulecatch{// phần xử lý những lỗi có thể sinh ra}
@member{// chứa các hàm hoặc biến toàn cục}
rulename: mô tả các luật sinh cho ngôn ngữ JavaMinus
```

Nội dung của gtype là rỗng đối với một file combined grammar. Khi muốn viết các file thuộc loại khác như lexer, parser, hay tree parser thì nội dung gtype sẽ thay đổi.

Các từ tố của ngôn ngữ được mô tả đầy dủ trong phần tokens{} của combined grammar trong ANTLR. Ví dụ các từ tố của ngôn ngữ Java được mô tả như sau:

```
Tokens{
	AND=‘&’;
	AND_ASIGN=‘&=’;
	ASSIGN=‘=’;
	COLON=‘:’;
	COMMA=‘,’;
	DEC=‘_ _’;
	DIV=‘/’;
	...
}
```

Ta sẽ sử dụng luật sinh để kết hợp với các tố từ này để sinh ra cú pháp cho ngôn ngữ. Các luật sinh này được viết trong phần rule của combined grammer. Cấu trúc chung của luật sinh ANTLR như sau:

```
rulename [args] returns [T val]
	@init{// những lệnh sẽ chạy trước khi parser một đoạn mã nguồn ứng với luật sinh này}
	@after{// những lệnh sẽ chạy sau khi parser một đoạn mã nguồn ứng với luật sinh này}
```

```
    : alternative1
    | alternative2
    | alternative3 ... ;
```

Trong cấu trúc này, args là các tham số đối với luật sinh, val là giá trị trả về của luật sinh. Các alternative là các luật sinh khác kết hợp với các từ tố.

Ví dụ một đoạn nhỏ luật sinh của ngôn ngữ Java:

```
program
    :   packageDeclaration?
        importDeclaration*
        typeDeclaration*
    ;
packageDeclaration
    :   'package' IDENT ('.'IDENT)* ';'
    ;
importDeclaration
    :   'import'  IDENT ('.'IDENT)* ';'
    ;
typeDeclaration
    :   classTypeDeclaration
    ;
classTypeDeclaration
    :   'class' IDENT '{'
        (variable|constant)*
        '}'
    ;
constant
    :    IDENT '=' INTEGER ';'
    ;
variable
    :   type IDENT ';'
    ;
type
    :   ('int'
        |'float'
        |'double'
        )
```

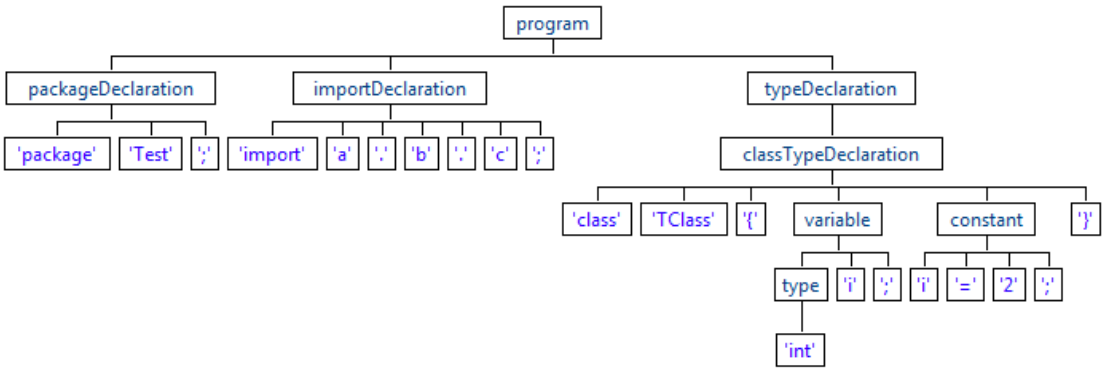
```
    ;
INTEGER : '0'..'9'+;
IDENT : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9')*;
WS: (' '|''|'\n'|'\r'|'\f')+ {$channel = HIDDEN;};
```

Luật sinh trên ANTLR hiểu là một đơn vị biên dịch trong java gồm có khai báo package, tiếp theo là khai báo import và sau đó đến các các loại khai báo class. Trong class sẽ khai báo biến và khai báo hằng.

Với một đoạn mã nguồn như sau:

```
package Test;
import a.b.c;
class TClass{
int i;
i = 2;
}
```

Áp dụng ví dụ luật sinh ở trên trình thông dịch sẽ cho ra cây cú pháp như sau:

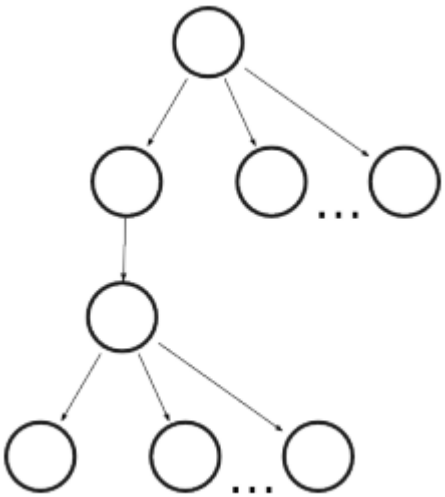


Hình 3.6 Ví dụ sinh cây cú pháp từ luật sinh đơn giản.

3.3 Tạo đồ thị dòng điều khiển cơ bản:

Để phục vụ cho mục đích tạo đồ thị dòng điều khiển cơ bản đơn giản hơn, ta sẽ duyệt cây cú pháp trừu tượng AST để xây dựng một mô hình cây thân thiện hơn cho phép ta dễ

dàng thao tác. Mô hình này là interface INode, mỗi node đại diện cho một lệnh và các node con chính là cấu trúc interface INode của đoạn lệnh bên dưới, hoặc bên trong. Các lệnh khác nhau sẽ có kiểu INode khác nhau.



Hình 3.7 Mô hình cây INode.

Cách tốt nhất để duyệt một AST đó là sử dụng ASTVisitor[9]. Các lớp con của ASTVisitor sẽ được truyền vào các node của AST. AST sẽ đệ quy từng bước dọc theo cây, và thực hiện các phương thức được đề xuất trong lớp con của ASTVisitor cho mỗi node của AST theo thứ tự (ví dụ cho một MethodInvocation):

- preVisit(ASTNode node).
- visit(MethodInvocation node).
- Các con của lời gọi phương thức được thực hiện đệ quy nếu kết quả trả về của hàm visit là true.
- endVisit(MethodInvocation node).
- postVisit(ASTNode node).

Trong chương trình class ASTNodeMainVisitor sẽ đảm nhiệm chức năng trên trên. Class ASTNodeMainVisitor thừa kế ASTVisitor và hiện thực phương thức visit các lệnh như sau:

Methods	Mô tả
visit(ExpressionStatement node)	Thêm một node vào mô hình cây nếu bắt gặp biểu thức. Trường hợp này bao gồm cả biểu thức điều kiện có dạng "expression ? expression : expression", nhưng được thêm vào cây như là một câu lệnh if.
visit(VariableDeclarationFragment node)	Thêm một node vào cây nếu gặp khai báo biến.
visit(IfStatement node)	Thêm một node vào cây nếu gặp câu lệnh if. Node này gồm ít nhất 3 node con. Node con thứ 2 sẽ chứa câu lệnh tiếp theo lệnh if. Nếu có khối lệnh else thì sẽ được thể hiện tại node thứ 4
visit(TryStatement node)	Thêm một node nếu gặp câu lệnh try. Node này sẽ chứa 4 node con.
visit(ForStatement node)	Thêm một node vào cây nếu gặp vòng lặp for. Node này gồm 5 node con. Node thứ 2 là câu lệnh sau vòng lặp for.
visit(WhileStatement node)	Thêm một node vào cây nếu gặp vòng lặp while. Node này gồm 3 node con. Node thứ 2 là câu lệnh sau vòng lặp while
visit(DoStatement node)	Thêm một node vào cây nếu gặp câu lệnh do-while. Điểm khác nhau giữa câu lệnh do-while với for hoặc while là, vòng lặp có thể được bỏ qua tại nút cuối mà không cần phải quan tâm đến nút đầu.
visit(SwitchStatement node)	Thêm một node vào cây nếu gặp câu lệnh switch. Node sẽ chứa tất cả các trường hợp case và default của switch.

visit(BreakStatement node)	Thêm một node vào cây nếu gặp câu lệnh break. Node này không chứa con nào.
visit(ContinueStatement node)	Thêm một node vào cây nếu gặp câu lệnh continue. Node này không chứa con nào.
visit(ReturnStatement node)	Thêm một node vào cây nếu gặp câu lệnh return. Node này không chứa con nào.

Bảng 3.1 Phương thức visit của visitor.

Sau khi hiện thực các phương thức visit trong ASTNodeMainVisitor cho từng lệnh Java, ta tiến hành duyệt cây AST và xây dựng mô hình cây mới như sau:

```
ASTVisitor visitor = new ASTNodeMainVisitor(targetPart, nodes);
ast.accept(visitor);
```

Trong đó ast chính là cây cú pháp trừu tượng AST được tạo ra từ bước một. Sau đó ta sẽ thu được một cấu trúc cây có node gốc là nodes.

Dựa vào mô hình cây được xây dựng ở trên, ta mới bắt đầu xây dựng đồ thị dòng điều khiển cơ bản. Vì tính chất của model nên chúng ta sẽ xây dựng trực tiếp đồ thị dòng điều khiển cơ bản, trong bước này câu lệnh switch được chuyển từ đa phân thành nhị phân, mỗi nút quyết định chứa nhiều điều kiện con được tách thành nhiều nút quyết định chỉ chứa một điều kiện. Thuật toán chính để xây dựng đồ thị dòng điều khiển cơ bản được hiện thực trong class TestcaseGraphBuilder. Class này cần 2 tham số chính đó là:

- Graph: chứa các node và các kết nối giữa các node, đối tượng này đại diện cho đồ thị dòng điều khiển cơ bản.
- Model: chính là mô hình cây mới sau khi duyệt AST đã được xây dựng ở trên.

TestcaseGraphBuilder sẽ xử lý model và lưu lại kết quả là một graph. Đối với mỗi kiểu node khác nhau TestcaseGraphBuilder sẽ xử lý khác nhau. Đầu tiên và dễ nhất để giải thích là node kiểu EXPRESSION_STATEMENT.

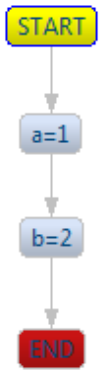
Expression statements:

Cấu trúc INode của lệnh này là node chứa một node con là node câu lệnh tiếp theo. Ví vậy ta chỉ cần tạo kết nối giữa node này và node con.

Ví dụ:

```
void expression_statement(int b,int a) {
	a = 1;
	b = 2;
}
```

Đồ thị được sinh ra như hình:



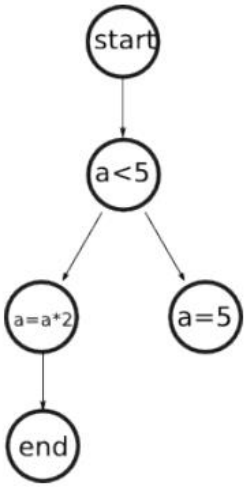
Hình 3.8 Đồ thị câu lệnh biểu thức.

If statements:

Điểm khác nhau giữa if statement và expression statement là câu lệnh if ngoài chứa một node con là node của câu lệnh tiếp theo còn chứa các node con là node chứa biểu thức điều kiện, các node các câu lệnh trong biểu thức then và node các câu lệnh trong khối else. Phụ thuộc vào sự tồn tại của câu lệnh else mà sẽ có kết nối giữa node biểu thức điều kiện với node câu lệnh tiếp theo trực tiếp hay không.

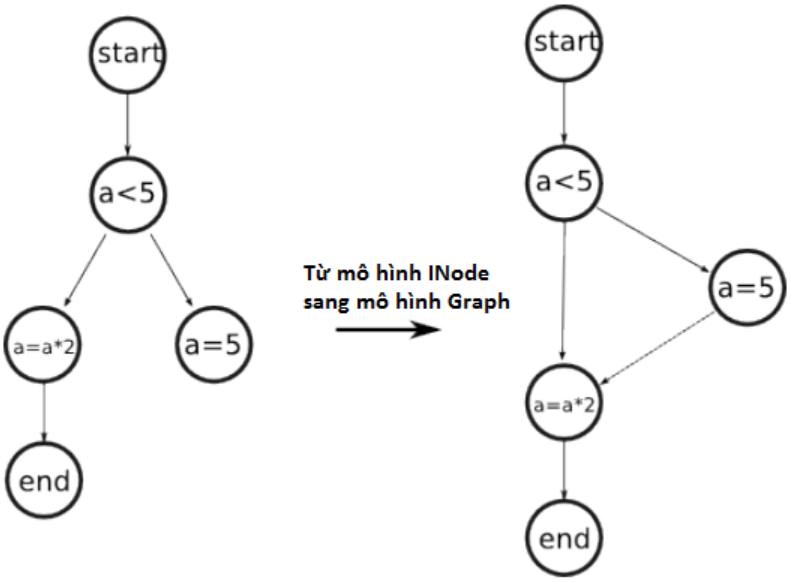
Ví dụ:

```
void ifTestMethod(int a) {
	if (a < 5) {
		a = 5;
	}
	a = a*2;
}
```



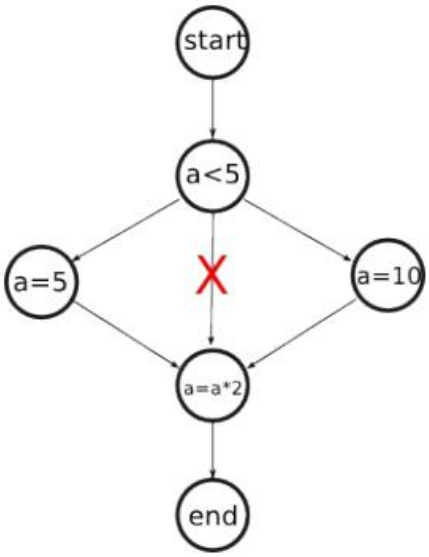
Hình 3.9 Thể hiện cấu trúc INode của câu lệnh if.

Tại node a = 5 của đồ thị trên nên có một kết nối với node a = a * 2. Vì vậy chúng ta sẽ tạo kết nối giữa chúng sau khi chuyển từ mô hình INode sang mô hình graph.



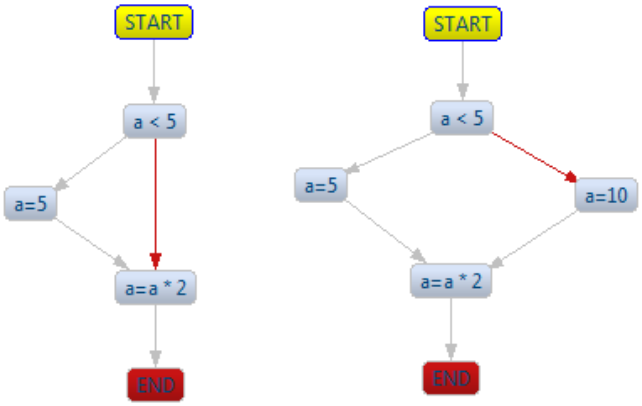
Hình 3.10 Chuyển từ INode sang Graph.

Nếu có câu lệnh else thì if-node sẽ có một con nữa và liên kết giữa node điều kiện và node của câu lệnh tiếp theo sẽ bị bỏ qua.



Hình 3.11 Mô hình graph của câu lệnh if-else.

Các trường hợp trên khi chạy chương trình được thể hiện như hình sau:



Hình 3.12 Câu lệnh if-else.

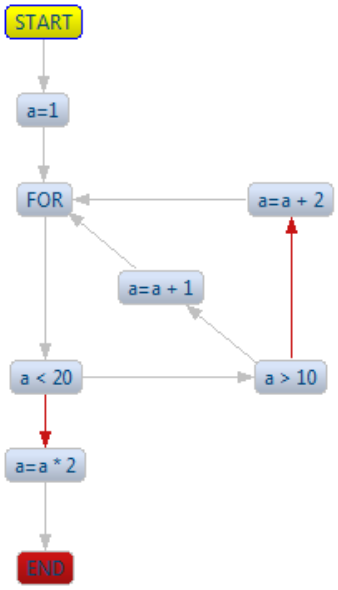
For statements:

Cấu trúc node này có chứa một node con là câu lệnh khởi tạo giá trị ban đầu của điều kiện, khi tạo graph node này sẽ tạo đầu tiên trước node kiểm tra điều kiện của vòng lặp, sau node kiểm tra điều kiện sẽ là node chứa cách lệnh con bên trong vòng lặp. Ngoài ra còn có trường hợp xuất hiện câu lệnh break hay là continue, đây là các node quyết định

xem nó nên kết nối với node đánh giá điều kiện (đối với lệnh continue) hay là kết nối với node của lệnh tiếp theo (đối với lệnh break) để thoát vòng lặp.

Ví dụ:

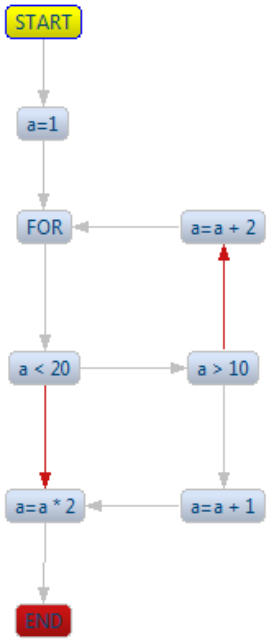
```
void forTestMethod(int a) {
	for (a = 1; a < 20;) {
		if (a > 10) {
			a = a + 1;
		} else {
			a = a + 2;
		}
	}
	a = a*2;
}
```



Hình 3.13 Vòng lặp for.

Chúng ta có thể thấy nếu điều kiện vòng lặp for a < 20 là false thì sẽ kết nối trực tiếp với câu lệnh tiếp theo a=a*2. Nếu a < 20 là true thì sẽ thực hiện câu lệnh if-else trong vòng lặp, trong trường hợp này thì cả hai nhánh if-else đều nối trở về node For.

Nếu chúng ta thêm bên dưới lệnh a=a + 1 một lệnh break, để hình dung tốt hơn ta cùng kiểm tra đồ thị được sinh ra khi chạy chương trình.



Hình 3.14 Vòng lặp for chứa câu lệnh break.

Ta có thể thấy nút điều kiện a > 10 của câu lệnh if nếu là true thì node a=a+1 sẽ nối với node câu lệnh tiếp theo a=a*2 thay vì nối trở lại node for.

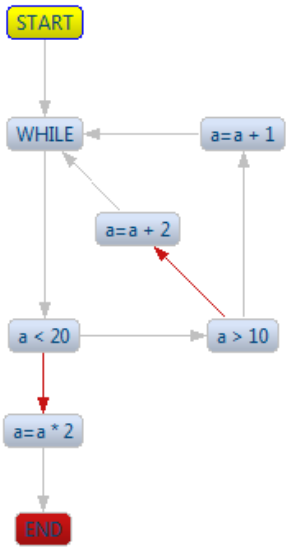
While statements:

Nếu hiểu theo mức logic thì câu lệnh while không khác câu lệnh for, điều kiện đều được kiểm tra trước khối lệnh trong vòng lặp, ngoại trừ câu lệnh for có node chứa câu lệnh khởi tạo giá trị ban đầu của phần kiểm tra điều kiện. Đối với trường hợp có chứa lệnh break hoặc continue, ta xử lý như với vòng lặp for.

Ví dụ:

```
void whileTestMethod(int a) {
	while (a < 20) {
		if (a > 10) {
			a = a + 1;
		} else {
			a = a + 2;
		}
	}
	a = a*2;
```

```
}
```



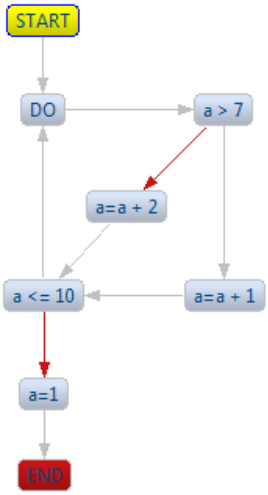
Hình 3.15 Vòng lặp while.

Do-while statements:

Câu lệnh Do-while cũng tương tự như câu lệnh while ngoại trừ ta thực hiện các câu lệnh bên trong trước khi kiểm tra biểu thức điều kiện. Về cơ bản biểu thức điều kiện sẽ ở dưới của vòng lặp và nó có thể bỏ qua từ cuối của khối mà không cần tham khảo lại bên trên. Để dễ hiểu hơn hãy xem ví dụ bên dưới:

```
void doWhileTestMethod(int a) {
	do {
		if (a > 7) {
			a = a + 1;
		} else {
			a = a + 2;
		}
	}while (a <= 10);
	a = 1;
}
```

Đồ thị ví dụ được thể hiện khi chạy chương trình như sau:



Hình 3.16 Vòng lặp do-while.

Ta sẽ hiện thực câu lệnh if trong nhánh Do trước khi là tiến hành kiểm tra điều kiện. Câu lệnh if sẽ được chạy ít nhất một lần và tại node kiểm tra điều kiện a<=10 nếu false sẽ nối trực tiếp với node câu lệnh tiếp theo mà không phải nối với node Do.

Switch-case statements:

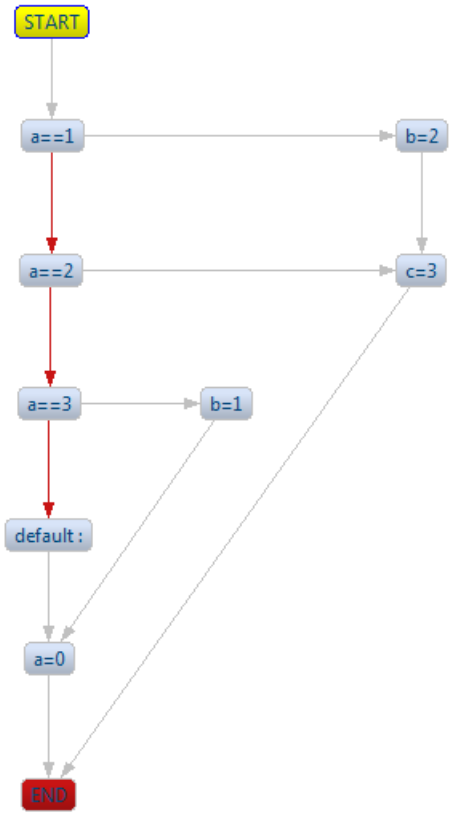
Nếu câu lệnh if chỉ cho phép một hoặc 2 lựa chọn thì câu lệnh switch-case cho phép số lượng không giới hạn lựa chọn. Vì tính chất của đồ thị dòng điều khiển cơ bản là chỉ chứa các nút quyết định nhị phân nhưng câu lệnh switch-case lại là đa phân, vì vậy trường hợp switch-case sẽ được xử lý để thành cấu trúc giống như là các lệnh if-else lồng nhau.

Ví dụ:

```
void switchTestMethod(int a, int b, int c) {
	switch (a) {
	case 1:
		b = 2;
	case 2:
		c = 3;
		break;
	case 3:
		b = 1;
	default:
		a = 0;
```

```
        }
    }
```

Câu lệnh trên khi được xử lý sẽ cho đồ thị như sau:



Hình 3.17 Câu lệnh switch-case.

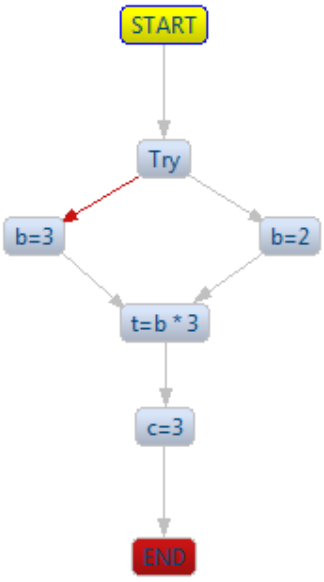
Với "case 1" trong lệnh switch được thể hiện như là câu lệnh if với điều kiện a==1, nhánh true sẽ là các lệnh được ghi bên trong câu lệnh "case 1" và nhánh false là "case 2", lúc này "case 2" sẽ được thể hiện như là một câu lệnh if khác với điều kiện a==2, và lệnh if này được lồng trong nhánh else của lệnh if trên. Các câu lệnh case khác của switch cũng được thực hiện với cơ chế tương tự. Để ý trong đoạn lệnh trên ta có chèn một câu lệnh break, nó sẽ kết thúc câu lệnh switch-case tạo kết nối giữa node câu lệnh bên trên lệnh break và node lệnh tiếp theo câu lệnh switch-case, ví dụ trên là tạo kết nối giữa node c=3 và node END.

Try-catch statements:

Câu lệnh này có cấu trúc node chứa các node con là khối lệnh catch và finally. Khối catch sẽ được thực thi khi xảy ra lỗi tại khối lệnh try, còn khối finally sẽ được thực thi dù khối lệnh try có được thực thi thành công hay không.

Ví dụ lệnh try-catch khi được thể hiện qua chương trình:

```
void tryCatchTestMethod(int b, int c, int t) {
	try {
		b = 2;
	}catch (Exception e) {
			b = 3;
	}finally {
			t = b*3;
		}
	c = 3;
}
```



Hình 3.18 Câu lệnh try-catch.

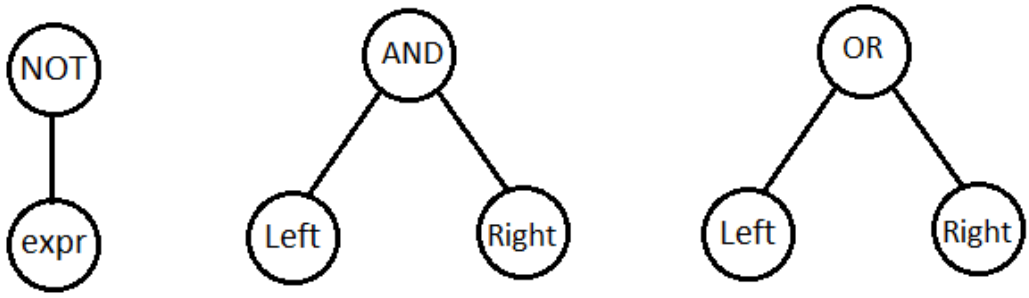
Xử lí tách điều kiện phức tạp thành các điều kiện con:

Để đạt mức độ kiểm thử phủ các nhánh và các điều kiện, ta cần phải tách riêng các điều con trong các điều kiện rẽ nhánh của câu lệnh if hay điều kiện lặp trong các câu lệnh có chứa vòng lặp (for, while và do-while).

Cấu trúc INode của các câu lệnh if, for, while và do-while đều có một node chứa cấu trúc LNode lưu thông tin của điều kiện phức tạp được tạo thành. Class ASTLogicalExpressionChecker sẽ thực hiện chức năng trên với phương thức visit_(Expression expr).

Các toán tử điều kiện phức tạp được xử lí bao gồm:

- Toán tử NOT (!) : Cấu trúc LNode được tạo ra gồm node cha chứa thông tin toán tử và một node con chứa thông tin biểu thức.
- Toán tử AND (&&, &) : Cấu trúc LNode được tạo ra gồm node cha chứa thông tin toán tử và hai node con chứa thông tin biểu thức bên trái và bên phải của toán tử.
- Toán tử OR (||, |): Cấu trúc LNode được tạo ra gồm node cha chứa thông tin toán tử và hai node con chứa thông tin biểu thức bên trái và bên phải của toán tử.



Hình 3.19: Cấu trúc LNode của các toán tử điều kiện.

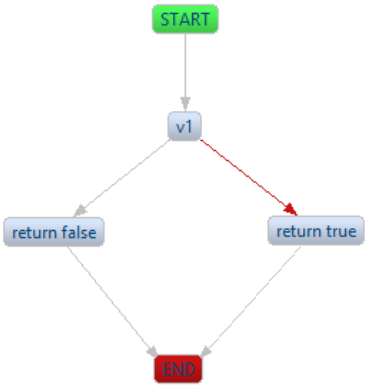
Với mỗi toán tử thì ta sẽ sinh ra đồ thị có dạng như các ví dụ sau:

- Toán tử NOT (Hình 3.17) :

```
boolean condition_not(boolean v1){
      if( !v1 )
            return true;
```

```
        else
                return false;
}
```

Câu lệnh trên khi được xử lý sẽ cho đồ thị như sau:

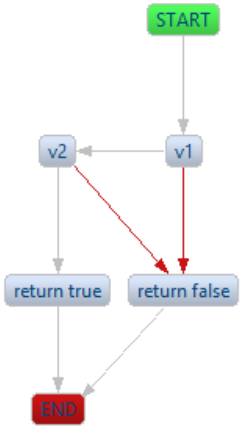


Hình 3.20: Điều kiện con với toán tử NOT.

- Toán tử AND (Hình 3.18):

```
boolean condition_and(boolean v1, boolean v2){
        if( v1 && v2)
                return true;
        else
                return false;
}
```

Câu lệnh trên khi được xử lý sẽ cho đồ thị như sau:

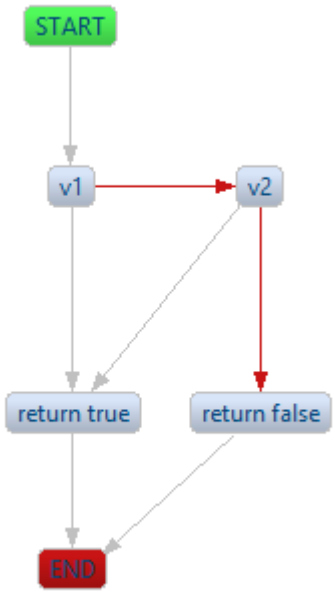


Hình 3.21: Điều kiện con với toán tử AND.

- Toán tử OR (Hình 3.19):

```
boolean condition_or(boolean v1, boolean v2){
	if( v1 || v2)
		return true;
	else
		return false;
}
```

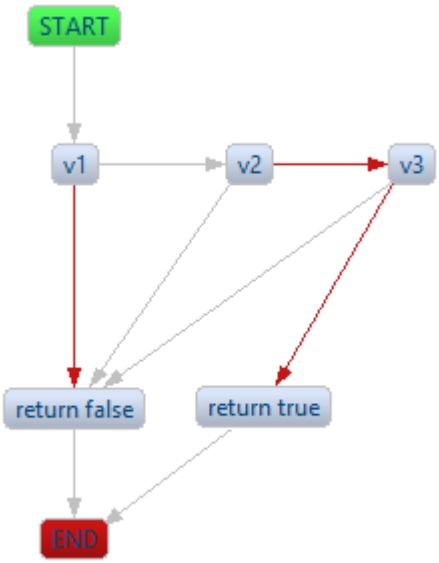
Câu lệnh trên khi được xử lý sẽ cho đồ thị như sau:



Hình 3.22: Điều kiện con với toán tử OR

- Biểu thức chứa cả ba toán tử NOT, AND, OR (Hình 3.20):

```
boolean condition_all(boolean v1, boolean v2, boolean v3){
	if( v1 && !(v2 || v3))
		return true;
	else
		return false;
}
```



Hình 3.23: Điều kiện con với các toán tử NOT, AND, OR

3.4 Xác định các đường thi hành tuyến tính độc lập và sinh testcase:

3.4.1 Xác định các đường thi hành tuyến tính độc lập:

Với các điều kiện của các đường thi hành tuyến tính độc lập đã nêu ở mục 2.4.4, dựa vào cấu trúc TestcaseGraph được sinh ra từ TestcaseGraphBuilder, ta xác định các đường thi hành tuyến tính độc lập bằng các tiêu chí sau:

- Tất cả các node đều phải được duyệt qua.
- Mỗi đường thi hành bắt đầu từ START và kết thúc ở END.
- Đường thi hành xác định từ một node gồm các nút đã duyệt từ nút START đến nút đó và các node tiếp theo được xác định bằng cách duyệt từ nút đó tới các nút con với điều kiện sau (điều kiện *):

  - Duyệt theo đường thi hành đúng khi gặp các node điều kiện trong câu lệnh if.
  - Thoát khỏi vòng lặp khi gặp các node điều kiện trong câu lệnh lặp (for, while, do-while).
  - Không duyệt tới các node đã duyệt trước đó.
  - Xác định được đường thi hành khi duyệt tới node END.

Đường thi hành đầu tiên là đường duyệt từ node START theo điều kiện (*) ở trên. Ở mỗi node nhị phân ta xác định được một đường thi hành từ node con chưa được duyệt. Tổng số đường thi hành tuyến tính độc lập bằng đúng số các node nhị phân cộng 1 cũng như theo công thức của Tom McCabe theo mục 2.4.5.

Với các tiêu chí trên chương trình được thực hiện theo các giải thuật sau :

- Giải thuật xác định đường thi hành từ một node theo điều kiện (*):

```
TestcaseNode[] getTruePath(TestcaseGraph g, TestcaseNode startNode, TestcaseNode endNode){
	TestcaseNode[] result;
	Stack stack;
```

```
        stack.push(startNode);
        while( stack is not empty){
                TestcaseNode n = stack.pop();
                if(n is endNode){
                        result.add(n);
                        return result;
                }
                else{
                        if(n has 1 child){
                                stack.push(child);
                                result.add(n);
                        }
                        else if(n has 2 child){
                                temp_chil = child is true in if condition or is out-loop in
                                                loop condition and result not contains child
                                stack.push(child);
                                result.add(n);
                        }

                }

        }
}
```

- Giải thuật duyệt qua tất cả các node để xác định đường thi hành từ các node nhị phân:

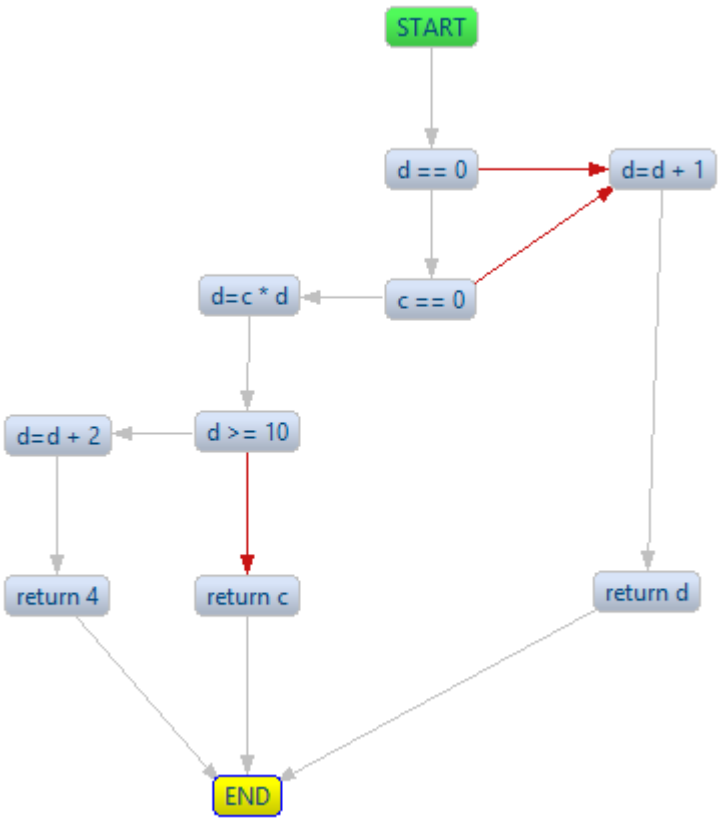
```
TestcaseNode[][] breadthFirstTraversal(TestcaseGraph g){
        TestcaseNode startNode = node START;
        TestcaseNode endNode = node END;
        result.add(getTruePath(g, startNode, endNode));
        Queue q;
        q.add(startNode);
        while(q is not empty){
                TestcaseNode n = q.poll();
                If(n has 1 child){
                        q.add(newTestcaseNode1);
                }
```

```
            else if(n has 2 child){
                  temp_child: child is not in another testcase.
                  TestcaseNode[] testcasePath = temp_child get before +
                        getTruePath(g, temp_child, endNode);
                  result.add(testcasePath);
                  if(child_1 is not visited) q.add(child_1);
                  if(child_2 is not visited) q.add(child_2);
            }
      }
      return result;
}
```

Ví dụ hàm chức năng chứa các câu lệnh if:

```
public int mutiple_if( int c, int d){
      if(!(d==0 && c==0)){
            d= d+1;
            return d;
      }
      else
            d = c*d;
      if(d>=10){
            d = d+2;
      }
      else
            return c;
      return 4;
}
```

Hàm chức năng trên sau khi xử lí được đồ thị dòng điều khiển như hình 3.21



Hình 3.24: Đồ thị dòng điều khiển hàm chức năng chứa các câu lệnh if.

Các đường thi hành tuyến tính độc lập được xác định gồm 4 đường:

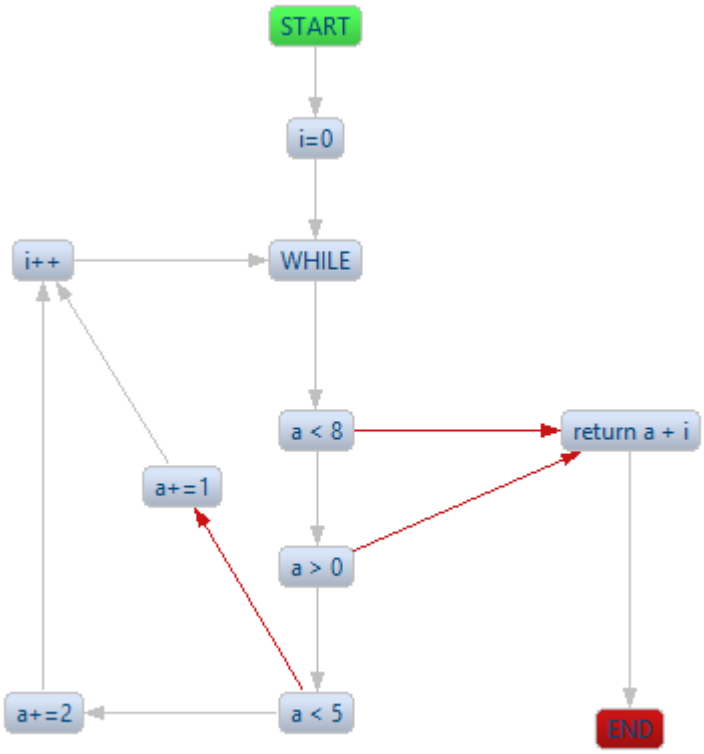
- [START, d == 0, c == 0, d=c * d, d >= 10, d=d + 2, return 4, END]
- [START, !(d == 0), d=d + 1, return d, END]
- [START, d == 0, c == 0, d=c * d, d >= 10, d=d + 2, return 4, END]
- [START, d == 0, c == 0, d=c * d, !(d >= 10), return c, END]

Ví dụ hàm chức năng chứa vòng lặp:

```
public int simple_while(int a){
	int i;
	i=0;
	while (a< 8 && a>0){
		if(a<5)
			a += 2;
		else
			a += 1;
```

```
            i++;
        }
        return a+i;
}
```

Hàm chức năng trên sau khi xử lí được đồ thị dòng điều khiển như hình 3.22



Hình 3.25: Đồ thị dòng điều khiển hàm chức năng chứa vòng lặp.

Các đường thi hành tuyến tính độc lập được xác định gồm 4 đường:

- [START, i=0, WHILE, !(a < 8), return a + i, END]
- [START, i=0, WHILE, a < 8, !(a > 0), return a + i, END]
- [START, i=0, WHILE, a < 8, a > 0, a < 5, a+=2, i++, WHILE, !(a < 8), return a + i, END]
- [START, i=0, WHILE, a < 8, a > 0, !(a < 5), a+=1, i++, WHILE, !(a < 8), return a + i, END]

3.4.2 Sinh testcase tự động cho mỗi đường thi hành tuyến tính độc lập:

Các bước tạo testcase cho đường thi hành tuyến tính độc lập gồm:

- Xác định các giá input đầu vào của hàm chức năng sao cho tất cả các câu lệnh trên đường thi hành đều thực thi thành công.
- Thực thi các câu lệnh trong đường thi hành với giá trị các giá trị input khởi tạo được xác định ở trên. Xác định kết quả trả về hoặc lỗi sinh ra sau khi thực thi.
- Sinh ra testcase theo cấu trúc JUnit.

Chi tiết các bước được trình bày dưới đây.

a) Xác định các giá trị input đầu vào:

Trên đường thi hành ở mỗi điều kiện con ta phải xác định được giá trị của các biến trong biểu thức để thỏa điều kiện con. Trong trường hợp này ta chỉ cần xác định một giá trị của mỗi biến thỏa mãn điều kiện con nên việc giải biểu thức để đưa ra các giá trị phù hợp là không cần thiết. Bên cạnh đó, chi phí để giải các biểu thức này rất lớn với các biểu thức phức tạp hoặc có thể không thể giải được bằng máy tính. Vì vậy, nhóm chỉ thực hiện được việc tính toán giá trị nhập vào có kiểu giá trị int trong một giới hạn giá trị do người thi hành chương trình nhập vào.

Với mỗi giới hạn giá trị của một tham số, chương trình sẽ thử lần lượt các giá trị tham số đầu vào từ giới hạn đó để thực thi các lệnh. Nếu tất cả các lệnh được thực thi thành công thì hoặc phát sinh thì sẽ xác định được giá trị trả về hoặc xác định được lỗi sinh ra với đường thi hành với tham số nhập vào trên.

b) Thực thi các lệnh trong đường thi hành:

Để thực thi các lệnh trên đường thi hành chương trình dùng Interpreter trong BeanShell[12].

Ví dụ thực thi các lệnh trong đường thi hành đường thi hành:

[START, !(d == 0), d=d + 1, return d, END]

```
Interpeter in = new Interpreter();
in.set(“d”,input);//input: parameter input
in.eval(“_con = !(d==0)”);
if( (Boolean) in.get(“_con”) == false ){
      return;//This input not right
```

```
}
in.eval(“d=d+1”);
in.eval(“return _result = d”);
outputExpected = in.get(“_result”);
```

Trong đường thi hành có các câu lệnh lặp thì chương trình sẽ thực hiện các lệnh trong vòng lặp nếu điều kiện lặp đúng cho đến khi câu lệnh thoát vòng lặp sau đó được thực hiện. Nếu các điều kiện lặp sai mà câu lệnh thoát vòng lặp chưa được thực hiện thì giá trị input nhập vào là không đúng cho đường thi hành này.

Ví dụ thực thi các lệnh trong đường thi hành:

[START, i=0, WHILE, !(a < 8), return a + i, END]

```
Interpeter in = new Interpreter();
in.set(“a”,input);//input: parameter input
ArrayList<TestcaseNode> newPath = [WHILE,!(a<8)];
If (TestcaseGenerator.runLoop([newPath, in, 2, outputExpected) == false){
      return;//This input not right
}
in.eval(“return _result = a+i);
outputExpected = in.get(“_result”);
```

Giải thuật thực thi các lệnh khi gặp vòng lặp:

```
boolean runLoop(Interpreter in, ArrayList<TestcaseNode> paths, int deep, String outputExpected){
      Interpreter new_in = copy of in;
      int gonext = 0;
      for(int i=0; i<paths.size(); i++){
            TestcaseNode test = paths.get(i);
            if ( test.Deep() > deep ) {
                  ArrayList<TestcaseNode> newpaths;
                  for(int j = i+1; j < paths.size(); j++){
                  TestcaseNode newtest = paths.get(j);
                  if(newtest.getDeep() > deep && newtest.getType is not loop ){
                        newpaths.add(newtest);
                  }
                  else{
```

```
                        i = j;
                        break;
                }
                return runLoop(new_in, newpaths, deep +1, outputExpected);
        }
        else{
                if(test.getType() is normal){
                        new_in.eval(test.getText());
                }
                else if(test.getType() is condition){
                        new_in.eval("_con = " + test.getText());
                        if(new_in.get("_con") == false){
                                gonext =1;
                                break;
                        }
                }
                else if(test.getType() is return){
                        new_in.eval return result;
                        outputExpected = new_in.get("_result”);
                }
        }
    }
    if(gonext == 0){
        in = new_in;
        return true;
    }
    else{
        if (in.eval loopConcondition == false {
                return false;
        }
        else{
                in.eval(loopBody);
                return runLoop(in, paths, deep, outputExpected);
        }
    }
}
```

c) Sinh ra testcase theo cấu trúc JUnit:

Một hàm chức năng khi sinh ra file testcase sẽ có thông tin sau:

- Chứa trong cùng thư mục với class chứa hàm chức năng.
- Tên file: <ClassName>_<MethodName>_<TypeSignature>_Test.java.

Nội dung trong file gồm:

- Package: cùng package với class chứa hàm chức năng;
- Import các class trong JUnit:

```
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.After;
```

- Khai báo class <ClassName>_<MethodName>_<TypeSignature>_Test gồm:
  - Khai báo biến myClass có kiểu của class chứa hàm chức năng.
  - Khai báo hàm khởi tạo:
    ```
    @Before public void initialize()
    ```
  - Khai báo hàm xử lí sau khi thực hiện testcase:
    ```
    @After public void clean()
    ```
  - Ứng với mỗi đường thi hành sẽ sinh ra được một hàm test:
    ```
    @Test public void test<numberOfTest>()
    ```

Để hiểu rõ hơn về file testcase được sinh ra và kết quả khi thực hiện file test case, người đọc có thể xem ở phần phụ lục 1 "Hướng sử dụng chương trình".

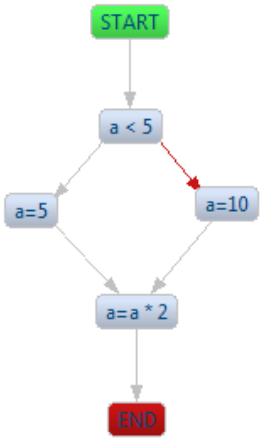
3.5 Chức năng hỗ trợ hiện thực thêm:

Để phục vụ cho quá trình tạo đồ thị dòng điều khiển được đơn giản và chính xác, chương trình có hiện thực thêm một chức năng gọi là "Build Graph" cho phép chúng ta có cái nhìn trực quan đồ thị dòng điều khiển được xây dựng. Chức năng này sẽ lấy kết quả sau khi tạo ra được đồ thị dòng điều khiển cơ bản và dùng bộ thư viện đồ họa zest để thể hiện các node, các kết nối thành các hình ảnh cụ thể.

Ta lấy lại ví dụ câu lệnh if-else ở trên:

```
void ifelseTestMethod(int a) {
	if (a < 5) {
		a = 5;
	}else
		a = 10;
	a = a*2;
}
```

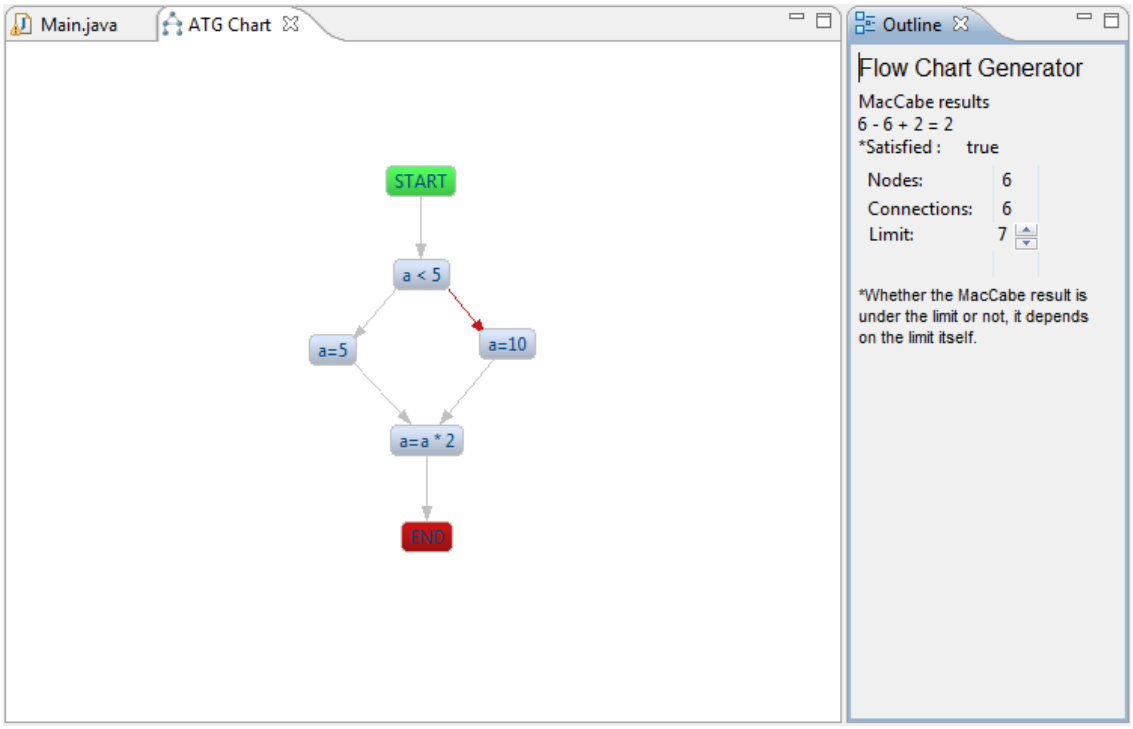
Sau khi sử dụng chức năng "Build Graph" ta nhận được đồ thị dưới:



Hình 3.26 Đồ thị biểu diễn bằng Build Graph

Quy ước node bắt đầu có màu xanh lá, node kết thức có màu đỏ và các node khác thì màu xanh dương. Các kết nối thể hiện nhánh false sẽ có màu đỏ, còn lại sẽ có màu xám.

Ngoài ra chức năng "Build Graph" còn tính giúp chúng ta số đường thi hành độc lập tuyến tính theo công thức McCabe V(G) = E – N + 2. Với ví dụ trên ta sẽ có 2 đường thi hành tuyến tính độc lập cơ bản:

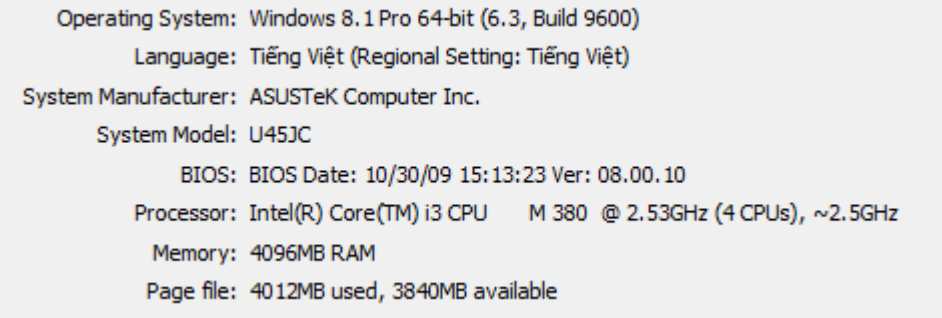


Hình 3.27 Chức năng tính đường thi hành độc lập

CHƯƠNG 4: THỰC NGHIỆM VÀ ĐÁNH GIÁ

4.1 Thực nghiệm:

Cấu hình máy tính thực hiện:



Trạng thái máy tính khi thực hiện: Chạy đồng thời các ứng dụng cơ bản khác như Google Chrome, Word, Excel.

Các hàm chức năng sử dụng để sinh testcase (Phụ lục 2: Các hàm chức năng sử dụng trong thực nghiệm). Các tham số đầu vào của các hàm chức năng đều thuộc kiểu *Int*. Thông tin của các hàm chức năng được miêu tả trong bảng sau:

Tên hàm	Tham số	Đường thi hành	Miêu tả
_fn01	0	1	Hàm đơn giản
_fn02	1	2	Hàm chứa câu lệnh if-else với điều kiện đơn giản
_fn03	1	3	Hàm chứa câu lệnh if-else liên tiếp với điều kiện đơn giản
_fn04	2	3	Hàm chứa câu lệnh if-else với điều kiện phức tạp (2 điều kiện con)
_fn05	3	5	Hàm chứa câu lệnh switch
_fn06	2	6	Hàm chứa các câu lệnh if-else lồng nhau với điều kiện đơn giản
_fn07	2	9	Hàm chứa các câu lệnh if-else lồng nhau với điều kiện đơn giản
_fn08	2	5	Hàm chứa câu lệnh switch, if-else
_fn09	2	4	Hàm chứa câu lệnh if-else liên tiếp, lồng nhau với điều kiện phức tạp (2 điều kiện con)
_fn10	3	4	Hàm chứa câu lệnh if-else với điều kiện phức tạp (3 điều kiện con)
_fn11	4	5	Hàm chứa câu lệnh if-else với điều kiện phức tạp (4 điều kiện con)
_fn12	1	4	Hàm chứa vòng lặp for có các câu lệnh if-else
_fn13	1	4	Hàm chứa vòng lặp while có các câu lệnh if-else
_fn14	1	4	Hàm chứa vòng lặp do-while có các câu lệnh if-else
_fn15	2	3	Hàm chứa 2 vòng lặp for lồng nhau
_fn16	2	5	Hàm chứa 2 vòng lặp while lồng nhau có các câu lệnh if-else

Bảng 6.1 Miêu tả các hàm sử dụng trong thực nghiệm

Kết quả thực nghiệm được thể hiện trong bảng sau:

STT	Hàm	Tham số	Độ sâu	Giới hạn dưới	Giới hạn trên	Đường thi hành	Thời gian (ms)
1	_fn01	0	1	-10	10	1	7
2	_fn02	1	1	-10	10	2	43
3	_fn02	1	1	-100	100	2	61
4	_fn02	1	1	-1000	1000	2	214
5	_fn03	1	1	-100	100	3	294
6	_fn03	1	1	-100	100	3	488
7	_fn03	1	1	-10	10	3	1214
8	_fn04	2	1	-10	10	3	1234
9	_fn04	2	1	-100	100	3	1464
10	_fn13	1	2	-10	10	4	1690
11	_fn15	0	2	-1000	1000	3	1757
12	_fn15	0	2	-100	100	3	1827
13	_fn15	0	2	-10	10	3	1872
14	_fn14	1	2	-10	10	4	1918
15	_fn12	1	2	-10	10	4	2229
16	_fn04	2	1	-1000	1000	3	2489
17	_fn05	1	1	-10	10	5	2507
18	_fn05	1	1	-10	10	5	2905
19	_fn05	1	1	-10	10	5	3953
20	_fn06	2	1	-1000	1000	6	5069
21	_fn14	1	2	-100	100	4	10551
22	_fn06	2	1	-1000	1000	6	10797
23	_fn13	1	2	-100	100	4	10874
24	_fn12	1	2	-100	100	4	11566
25	_fn10	3	1	-10	10	4	26213
26	_fn06	2	1	-10	10	6	42313

27	_fn07	2	1	-100	100	9	48153
28	_fn07	2	1	-100	100	9	92820
29	_fn12	1	2	-1000	1000	4	97903
30	_fn14	1	2	-1000	1000	4	98488
31	_fn13	1	2	-1000	1000	4	99919
32	_fn07	2	1	-100	100	9	143427
33	_fn08	2	1	-100	100	5	192453
34	_fn16	2	2	-10	10	5	261650
35	_fn08	2	1	-100	100	5	366673
36	_fn11	4	1	-10	10	5	501937
37	_fn08	2	1	-100	100	5	FAIL
38	_fn09	2	1	-1000	1000	4	FAIL
39	_fn09	2	1	-100	100	4	FAIL
40	_fn09	2	1	-1000	1000	4	FAIL
41	_fn10	3	1	-100	100	4	FAIL
42	_fn16	2	2	-100	100	5	FAIL

Bảng 6.1 Kết quả thực nghiệm

4.2 Đánh giá:

Dựa vào kết quả thực nghiệm trên, ta có một số đánh giá:

- Testcase sinh ra đúng theo các đường thi hành tuyến tính độc lập.
- Kết quả trả về do chương trình tính được theo mỗi đường thi hành đúng chính xác với kết quả khi thực hiện hàm với các testcase được sinh ra.
- Thời gian thực hiện việc tạo testcase biến thiên theo độ số tham số, giới hạn các tham số, độ sâu của vòng lặp cũng như độ phức tạp của các điều kiện trong các câu lệnh điều khiển dòng dữ liệu.
- Chương trình không thể xác định giá trị input trong giới hạn thời gian nhất định cho các hàm chức năng chứa nhiều vòng lặp lồng nhau, giới hạn giá trị của tham số quá lớn, có nhiều tham số với các điều kiện phức tạp.

CHƯƠNG 5: KẾT QUẢ ĐẠT ĐƯỢC VÀ HƯỚNG PHÁT TRIỂN

5.1 Kết quả đạt được:

Tìm hiểu về cơ chế kiểm thử luồng điều khiển, cụ thể là kiểm thử hộp trắng. Theo đó, chúng ta cần xây dựng được đồ thị dòng điều khiển cơ bản, xác định tập các đường thi hành tuyến tính độc lập cơ bản theo thuật giải đã trình bày. Dựa trên tập các đường thi hành tuyến tính độc lập xây dựng testcase cho từng đường thi hành này.

Để xây dựng đồ thị dòng điều khiển cơ bản, chúng ta có thể sử dụng công cụ ANTLR có sẵn. Công cụ này cho phép ta phân tích bộ từ vựng và cú pháp, giúp xây dựng cây cú pháp cho mã nguồn của bất kỳ ngôn ngữ nào dựa vào các luật sinh được viết trong file combined grammar. ANTLR cũng hỗ trợ ta Tree Grammar giúp duyệt qua cây cú pháp, đồng thời viết thêm các hành động ANTLR cho các luật sinh để xử lý ngữ nghĩa của các nút trong cây cú pháp và tạo ra một cấu trúc đồ thị có hướng mô tả đồ thị dòng điều khiển cơ bản. Dựa trên cơ chế này, chúng ta có thể tiến hành kiểm thử cho bất kỳ một ngôn ngữ nào.

Trong phạm vi của báo này, Nhóm không sử dụng ANTLR để sinh ra cây cú pháp trừu tượng, mà sử dụng AST của Eclipse Java Development Tools (JDT) để truy xuất mã nguồn Java. Sau đó sử dụng các phương thức mà nhóm đã xây dựng để tạo đồ thị dòng điều khiển cơ bản. Từ đồ thị dòng điều khiển cơ bản, nhóm thực hiện các giải thuật để sinh ra được các đường thi hành tuyến tính độc lập và tự động tạo testcase cho tất cả các đường thi hành của hàm chức năng.

Xây dựng chức năng hiển thị đồ thị dòng điều khiển cơ bản một cách trực quan trên giao diện của Eclipse IDE. Đồng thời, tính được tổng số đường thi hành tuyến độc lập dựa theo công thức của McCabe.

5.2 Hướng phát triển:

Ở chương trình mà nhóm phát triển, giải thuật sinh sinh giá trị đầu vào cho các tham số chỉ mới thực hiện được với tham số kiểu int với điều kiện giá trị giới hạn do người sử dụng chương trình nhập vào. Bên cạnh đó, hiệu suất thực thi các lệnh trong đường thi

hành tuyến tính độc lập để sinh ra giá trị trả về mong muốn còn thấp, dẫn đến hàm chức năng có nhiều cấu trúc lặp có thể không sinh ra được testcase như mong muốn.

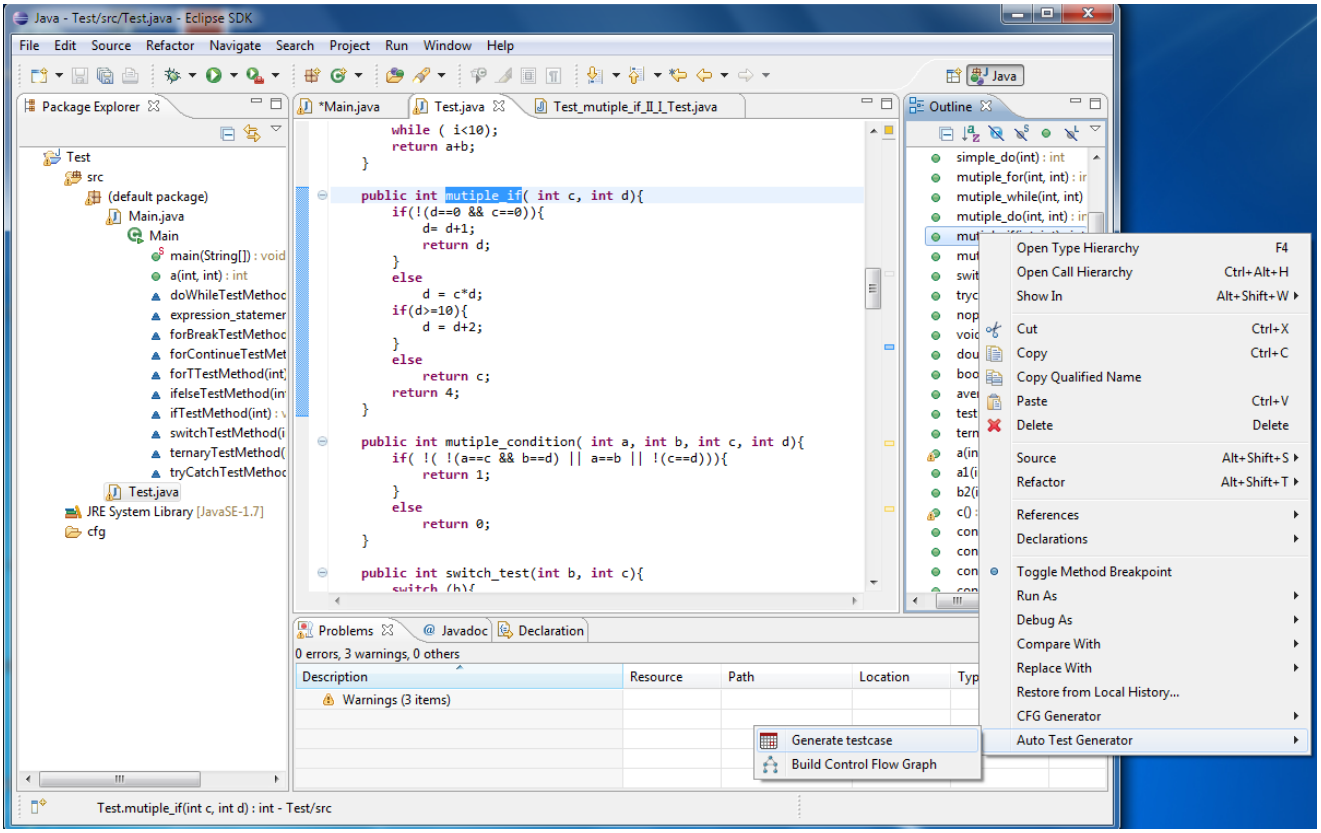
Vì vậy, hướng phát triển của để tài là tối ưu lại các giải thuật sinh testcase.

TÀI LIỆU THAM KHẢO

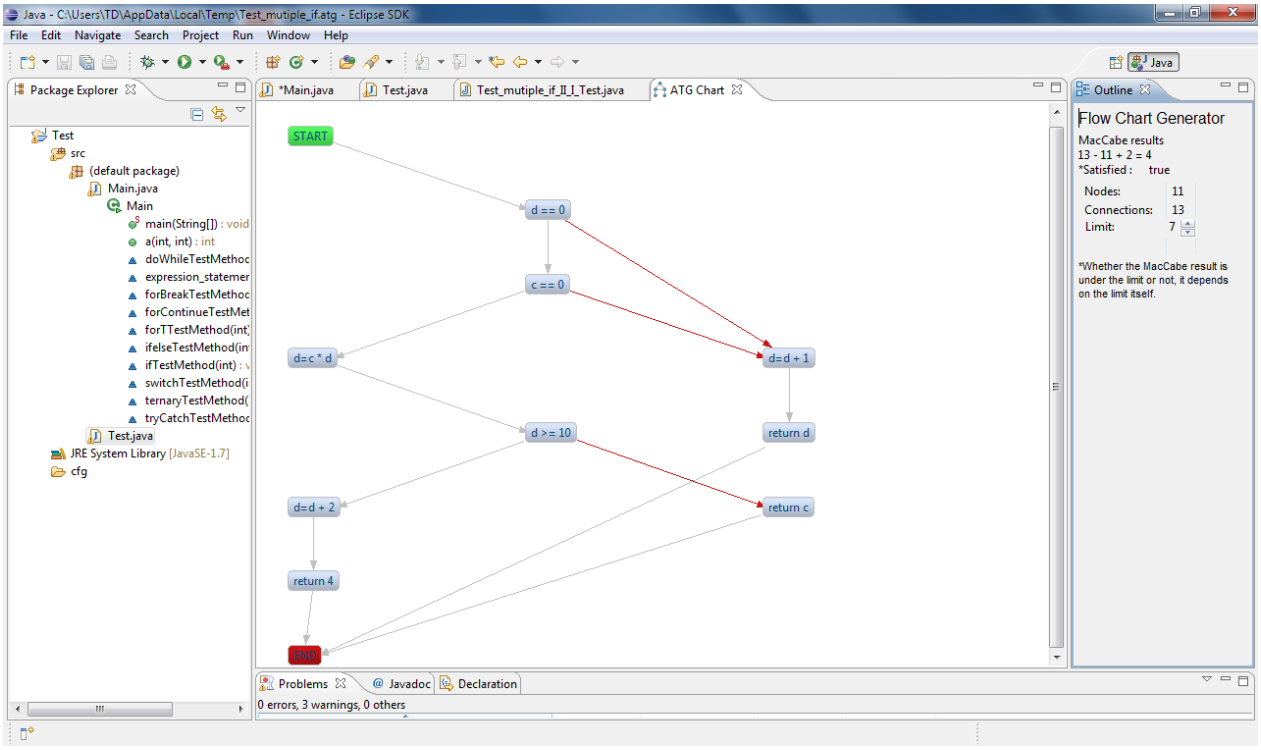
1. Slide bài giảng kiểm thử phần mềm, Nguyễn Văn Hiệp [https://cse.hcmut.edu.vn/~hiep/KiemthuPhanmem/]
2. Abstract syntax tree [http://en.wikipedia.org/wiki/Abstract_syntax_tree]
3. Eclipse Application [http://www.eclipse.org/]
4. Eclipse SDK API [http://help.eclipse.org/]
5. Eclipse GEF [http://www.eclipse.org/gef/]
6. GEF Tutorial [http://www.ibm.com/developerworks/library/os-eclipse-gef11/]
7. Eclipse Development [http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf]
8. ANTLR Tools [http://www.antlr.org/]
9. Abstract Syntax Tree [http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html]
10. Control Flow Graph Generator, Aldi Alimucaj, June 2009
11. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, Arthur H. Watson Thomas J. McCabe
12. BeanShell [http://www.beanshell.org/docs.html]

PHỤ LỤC 1 : HƯỚNG DẪN SỬ DỤNG CHƯƠNG TRÌNH

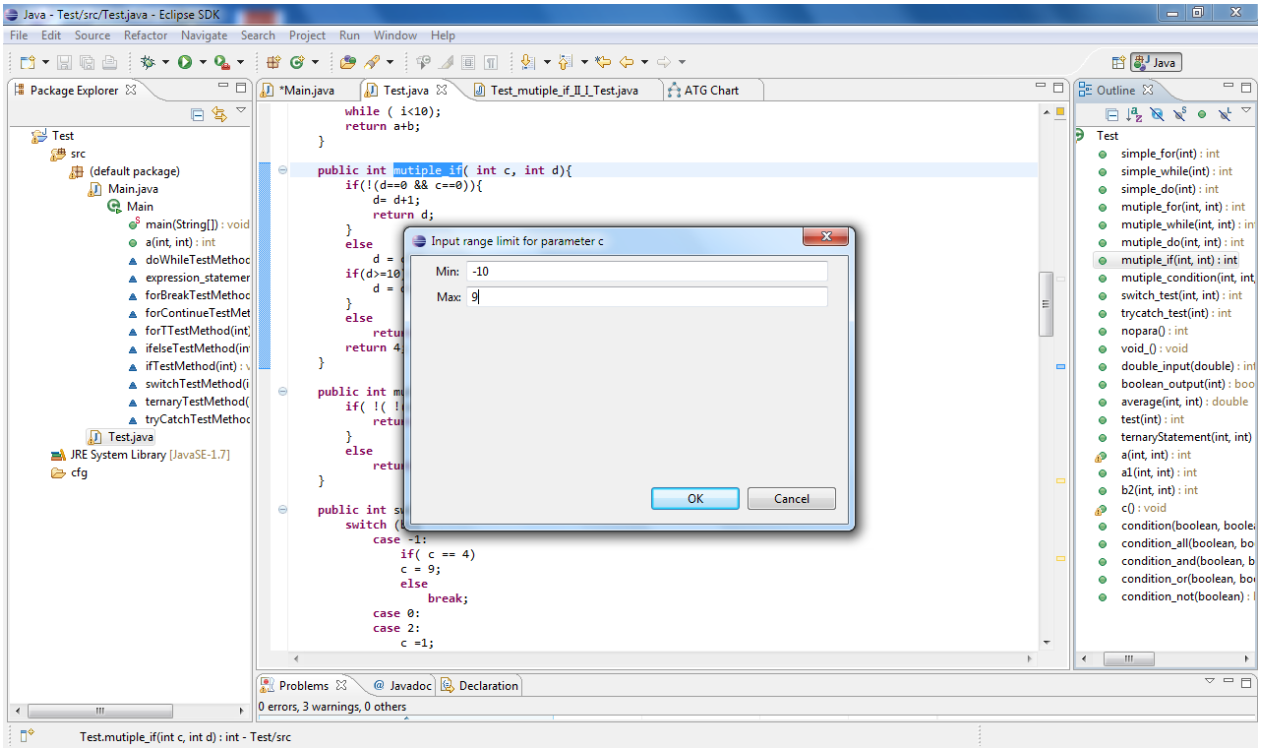
Sau khi plugin được cài đặt thì khi nhấp chuột phải vào một hàm chức năng ở Outline thì sẽ có Menu Auto Test Generator hiện ra như hình dưới:



Khi chọn menu Auto Test Generator -> Build Control Flow Graph thì ta sẽ được đồ thị như hình:



Khi Chọn Menu Auto Test Generator -> Generate Testcase. Chương trình mời người dùng nhập cái giá trị giới hạn cho các tham số:



Với hàm mutilple_if ở hình trên thì file testcase được sinh ra là

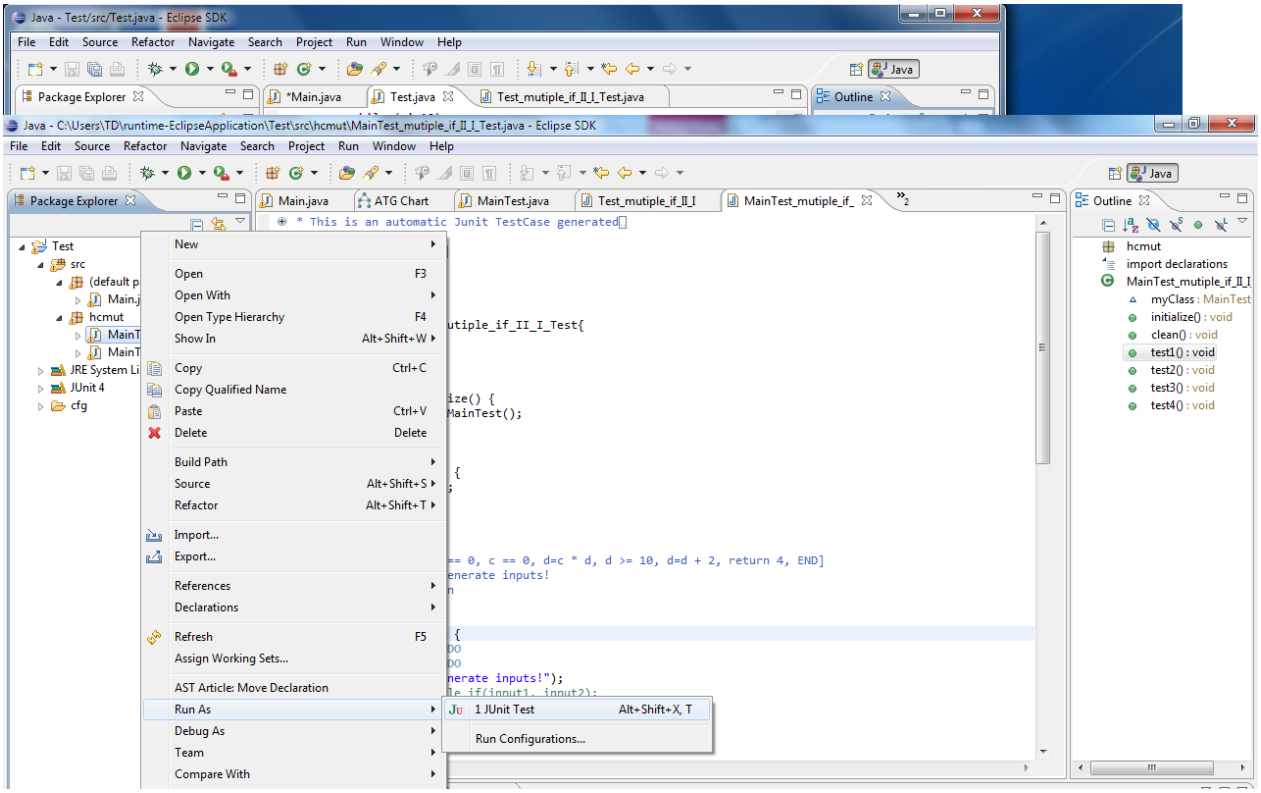
MainTest_mutiple_if_II_I_Test.java

```
/**
 * This is an automatic Junit TestCase generated
 * by ATG.
 */
package hcmut;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.After;

/*
 * @author ATG
 */
public class MainTest_mutiple_if_II_I_Test{

      MainTest myClass;
```

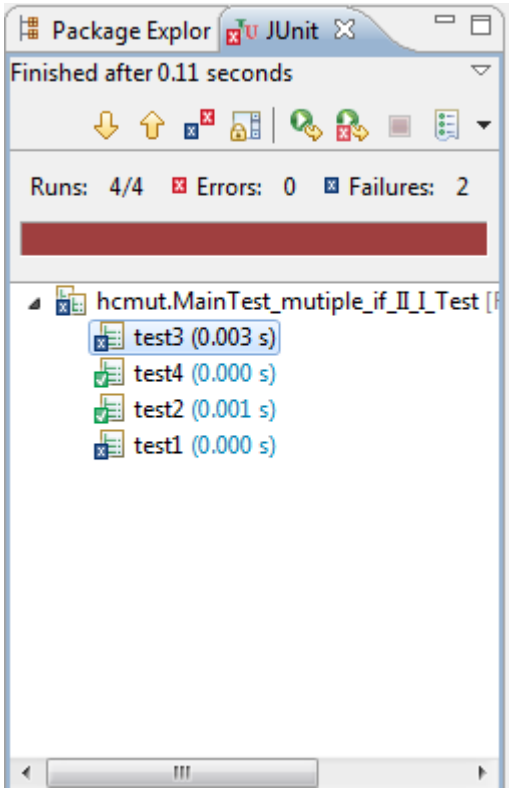
```
@Before
public void initialize() {
        myClass = new MainTest();
}

@After
public void clean() {
        myClass = null;
}

/**
 * Test Number 1
 * Path: [START, d == 0, c == 0, d=c * d, d >= 10, d=d + 2, return 4, END]
 * Result: Cannot generate inputs!
 * @throws Exception
 */
@Test
public void test1() {
       int input1;//TODO
       int input2;//TODO
       fail("Cannot generate inputs!");
       //myClass.mutiple_if(input1, input2);
}

/**
 * Test Number 2
 * Path: [START, !(d == 0), d=d + 1, return d, END]
 * Result: Ok
 * @throws Exception
 */
@Test
public void test2() {
       int input1 = -10;
       int input2 = -10;
       int output = myClass.mutiple_if(input1, input2);
       int outputExpected = -9;
       //compare results
```

```
		assertEquals("The result expected",outputExpected,output);
	}

	/**
	 * Test Number 3
	 * Path: [START, d == 0, c == 0, d=c * d, d >= 10, d=d + 2, return 4, END]
	 * Result: Cannot generate inputs!
	 * @throws Exception
	 */
	@Test
	public void test3() {
		int input1;//TODO
		int input2;//TODO
		fail("Cannot generate inputs!");
		//myClass.mutiple_if(input1, input2);
	}

	/**
	 * Test Number 4
	 * Path: [START, d == 0, c == 0, d=c * d, !(d >= 10), return c, END]
	 * Result: Ok
	 * @throws Exception
	 */
	@Test
	public void test4() {
		int input1 = 0;
		int input2 = 0;
		int output = myClass.mutiple_if(input1, input2);
		int outputExpected = 0;
		//compare results
		assertEquals("The result expected",outputExpected,output);
	}
}
```

Nhấp chuột phải vào file testcase được tạo ra chọn Run as -> JUnit



Kết quả sau khi chạy testcase bằng JUnit:



PHỤ LỤC 2: CÁC HÀM CHỨC NĂNG SỬ DỤNG TRONG THỰC NGHIỆM

```
public int _fn01(){
	int a = 4;
	return a;
}

public int _fn02(int a){
	if(a==0){
		return 1/a;
	}
	else
		return a;
}

public int _fn03(int a){
	if(a<0){
		return a;
	}
	else if(a==0){
		return 1/a;
	}
	else
		return a;
}

public int _fn04( int c, int d){
	if(!(d==0 && c==6)){
		d= d+1;
		return d;
	}
	else
		return c;
}

public void _fn05(int a, int x, int y){
	switch (a){
		case 1:
			System.out.println("Enter the number one=" + (x+y));
			break;
		case 2:
			System.out.println("Enter the number two=" + (x-y));
			break;
		case 3:
			System.out.println("Enetr the number three="+ (x*y));
			break;
		case 4:
			System.out.println("Enter the number four="+ (x/y));
			break;
		default:
			System.out.println("Invalid Entry!");
	}
}

public int _fn06( int a, int b){
	if(a<0){
		if(b<0){
```

```
                return 1;
            }
            else{
                return 2;
            }
        }
        else if(a==0){
            if(b<0){
                return 3;
            }
            else{
                return 4;
            }
        }
        else{
            if(b<0){
                return 5;
            }
            else{
                return 6;
            }
        }
    }
    public int _fn07( int a, int b){
        if(a<0){
            if(b<0){
                return 1;
            }
            else if (b==0){
                return 2;
            }
            else{
                return 3;
            }
        }
        else if(a==0){
            if(b<0){
                return 4;
            }
            else if (b==0){
                return 5;
            }
            else{
                return 6;
            }
        }
        else{
            if(b<0){
                return 7;
            }
            else if (b==0){
                return 8;
            }
            else{
                return 9;
            }
        }
    }
```

```
public int _fn08(int b, int c){
	switch (b){
		case -1:
			if( c == 4)
			c = 9;
			else
				break;
		case 0:
		case 2:
			c =1;
			break;
		default:
			b = 3;
			break;
	}
	return b;
}

public int _fn09( int c, int d){
	if(!(d==0 && c==6)){
		d= d+1;
		return d;
	}
	else
		d = c;
	if(d>=5){
		d = d+2;
	}
	else
		return c;
	return 4;
}

public int _fn10( int a, int b, int c){
	if( !( !(a==0 && b==3) || c==1 )){
		return 1;
	}
	else
		return 0;
}

public int _fn11( int a, int b, int c, int d){
	if( !( !(a==0 && b==3) || c==1 || !(d>0))){
		return 1;
	}
	else
		return 0;
}

public int _fn12(int a){
	int i;
	for (i=0; a< 8 && a>0; i++){
		if(a<5)
			a += 2;
		else
			a += 1;
	}
	return a+i;
}
```

```
public int _fn13(int a){
	int i;
	i=0;
	while (a< 8 && a>0){
		if(a<5)
			a += 2;
		else
			a += 1;
		i++;
	}
	return a+i;
}

public int _fn14(int a){
	int i;
	i=0;
	do{
		if(a<5)
			a += 2;
		else
			a += 1;
		i++;
	}
	while (a< 8 && a>0);
	return a+i;
}

public int _fn15(int a, int b){
	int i;
	int j;
	for (i=0; i<3; i++){
		for (j=0; j <3; j++){
			a +=1;
			b +=1;
		}
	}
	return a+b;
}

public int _fn16(int a, int b){
	int i;
	int j;
	i=0;
	while (a>3 && a< 5){
		j=0;
		while (b>0 && b<3){
			a +=1;
			b +=1;
			j++;
		}
		i = i+j;
	}
	return i;
}
```