Các kỹ thuật kiểm thử đột biến và ứng dụng kiểm thử chương trình C

Nguyễn Thị Miên

Trường Đại học Khoa học Tự nhiên
Khoa Toán - Cơ - Tin học
Chuyên ngành: Bảo đảm toán học cho máy tính và hệ thống tính toán; Mã số: 60.46.35
Người hướng dẫn: PGS. TS. Đoàn Văn Ban
Năm bảo vệ: 2011

Abstract. Trình bày khái quát về kiểm thử phần mềm như: khái niệm kiểm thử phần mềm, mục đích, mục tiêu và các mức kiểm thử phần mềm. Đề cập đến việc sử dụng các kỹ thuật kiểm thử hộp trắng và hộp đen để thiết kế dữ liệu thử. Mô tả chi tiết các thành phần chính của kỹ thuật kiểm thử đột biến, giới thiệu các giả thuyết cơ bản cần thiết để thực hiện phương pháp này. Tìm hiểu quy trình để phân tích đột biến, từ đó rút ra được những vấn đề còn hạn chế đối với kỹ thuật kiểm thử đột biến. Giới thiệu một số phương pháp cải tiến kỹ thuật kiểm thử đột biến nhằm giảm chi phí tính toán và tăng tự động hóa. Tập trung vào ứng dụng kỹ thuật kiểm thử đột biến. Giới thiệu hai công cụ mã nguồn mở miễn phí là NUnit dùng để kiểm thử đơn vị của chương trình C#, và Nester với chức năng phân tích và tạo đột biến. Ứng dụng kỹ thuật kiểm thử đột biến để kiểm thử các chương trình C# sử dụng hai công cụ trên

Keywords. Tin học; Kiểm thử đột biến; Kiểm thử chương trình C; Phần mềm

Content.

Kiểm thử phần mềm là một hoạt động giữ vai trò rất quan trọng để bảo đảm chất lượng phần mềm và là hoạt động mang tính sống còn trong các dự án sản xuất hoặc gia công phần mềm. Vì vậy, kiểm thử phần mềm đã trở thành qui trình bắt buộc trong các dự án phát triển phần mềm trên thế giới. Ở Việt Nam, ngành công nghiệp phần mềm đang phát triển thì không thể xem nhẹ việc kiểm thử phần mềm vì xác suất thất bại sẽ rất cao, hơn nữa, hầu hết các công ty phần mềm có uy tín đều đặt ra yêu cầu nghiêm ngặt là nếu một phần mềm không có tài liệu kiểm thử đi kèm thì sẽ không được chấp nhận.

Tuy nhiên, hoạt động kiểm thử thường gặp nhiều khó khăn:

- Thứ nhất, kiểm thử các hệ thống phức tạp đòi hỏi rất nhiều nguồn tài nguyên và chi phí cao.
- Thứ hai, tiến trình phát triển phần mềm luôn trải qua nhiều hoạt động biến đổi thông tin, sự mất mát thông tin trong quá trình biến đổi là yếu tố chính làm cho hoạt động kiểm thử khó khăn.
- Thứ ba, kiểm thử chưa được chú trọng trong đào tạo con người.
- Cuối cùng, không tồn tại kỹ thuật kiểm thử cho phép khẳng định một phần mềm hoàn toàn đúng đắn hay không chứa lỗi.

Với mục đích phát hiện lỗi, kiểm thử phần mềm thường phải trải qua các bước: tạo dữ liệu thử, thực thi phần mềm trên dữ liệu thử và quan sát kết quả nhận được. Trong các bước này, bước tạo dữ liệu đóng vai trò quan trọng nhất, bởi vì chúng ta không thể tạo ra mọi dữ liệu từ miền vào của chương trình, mà chúng ta chỉ có thể tạo ra các dữ liệu thử có khả năng phát hiện lỗi cao nhất. Vấn đề đặt ra là làm thế nào để đánh giá được khả năng phát hiện lỗi của một bộ dữ liệu thử?

Một kinh nghiệm để giúp giải quyết vấn đề này, đó là sử dụng khái niệm *chất lượng bộ dữ liệu thử* như là một phương tiện để đánh giá bộ dữ liệu thử như thế nào là "tốt" khi kiểm thử chương trình. Ở đây, "tốt" được đánh giá liên quan đến tiêu chuẩn chất lượng được định trước, thường là một số dấu hiệu bao phủ chương trình. Ví dụ, tiêu chuẩn bao phủ dòng lệnh đòi hỏi bộ dữ liệu thử thực hiện mọi dòng lệnh trong chương trình ít nhất một lần. Nếu bộ dữ liệu thử được tìm thấy không chất lượng liên quan đến tiêu chuẩn (tức là không phải tất cả các câu lệnh đều được thực hiện ít nhất một lần), thì kiểm thử nữa là bắt buộc. Do đó, mục tiêu là tạo ra một tập các kiểm thử thực hiện đầy đủ tiêu chuẩn chất lượng.

Tiêu chuẩn chất lượng tiêu biểu như bao phủ câu lệnh và kiểm thử quyết định (thực hiện tất cả các đường dẫn đúng và sai qua chương trình) dựa vào việc thực hiện chương trình với số lượng kiểm thử tăng dần để nâng cao độ tin cậy của chương trình đó. Tuy nhiên, chúng không tập trung vào nguyên

nhân thất bại của chương trình - được gọi là lỗi. Kiểm thử đột biến là một tiêu chuẩn như vậy. Tiêu chuẩn này tạo ra các phiên bản của chương trình có chứa các lỗi đơn giản và sau đó tìm ra các kiểm thử để chỉ ra các dấu hiệu của lỗi. Nếu có thể tìm thấy một bộ dữ liệu thử chất lượng làm lộ ra các dấu hiệu này ở tất cả các phiên bản bị lỗi, thì sự tin tưởng vào tính đúng đắn của chương trình sẽ tăng. Kiểm thử đột biến đã được áp dụng cho nhiều ngôn ngữ lập trình như là một kỹ thuật kiểm thử hộp trắng.

Ý thức được đây là một lĩnh vực nghiên cứu có nhiều triển vọng ứng dụng trong phát triển phần mềm, tôi đã chọn hướng nghiên cứu **" *Các kỹ thuật kiểm thử đột biến và ứng dụng kiểm thử chương trình C"*** cho đề tài luận văn của mình.

Luận văn được tổ chức thành 4 chương như sau:

- **Chương 1 –** Trình bày khái quát về kiểm thử phần mềm như khái niệm kiểm thử phần mềm, mục đích, mục tiêu và các mức kiểm thử phần mềm. Chương này cũng đề cập đến việc sử dụng các kỹ thuật kiểm thử hộp trắng và hộp đen để thiết kế dữ liệu thử.
- **Chương 2 -** Mô tả chi tiết các thành phần chính của kỹ thuật kiểm thử đột biến, giới thiệu các giả thuyết cơ bản cần thiết để thực hiện phương pháp này. Chương này còn cung cấp quy trình để phân tích đột biến, từ đó rút ra được những vấn đề còn hạn chế đối với kỹ thuật kiểm thử đột biến, được cải tiến ở chương 3.
- **Chương 3 –** Giới thiệu một số phương pháp cải tiến kỹ thuật kiểm thử đột biến nhằm giảm chi phí tính toán và tăng tự động hóa.
- **Chương 4 –** Tập trung vào ứng dụng kỹ thuật kiểm thử đột biến. Phần đầu giới thiệu hai công cụ mã nguồn mở miễn phí là NUnit dùng để kiểm thử đơn vị của chương trình C#, và Nester với chức năng phân tích và tạo đột biến. Tiếp đó là ứng dụng kỹ thuật kiểm thử đột biến để kiểm thử các chương trình C# sử dụng hai công cụ trên.

CHƯƠNG 1 – KHÁI QUÁT VỀ KIỂM THỬ PHẦN MỀM

1.1. Khái niệm

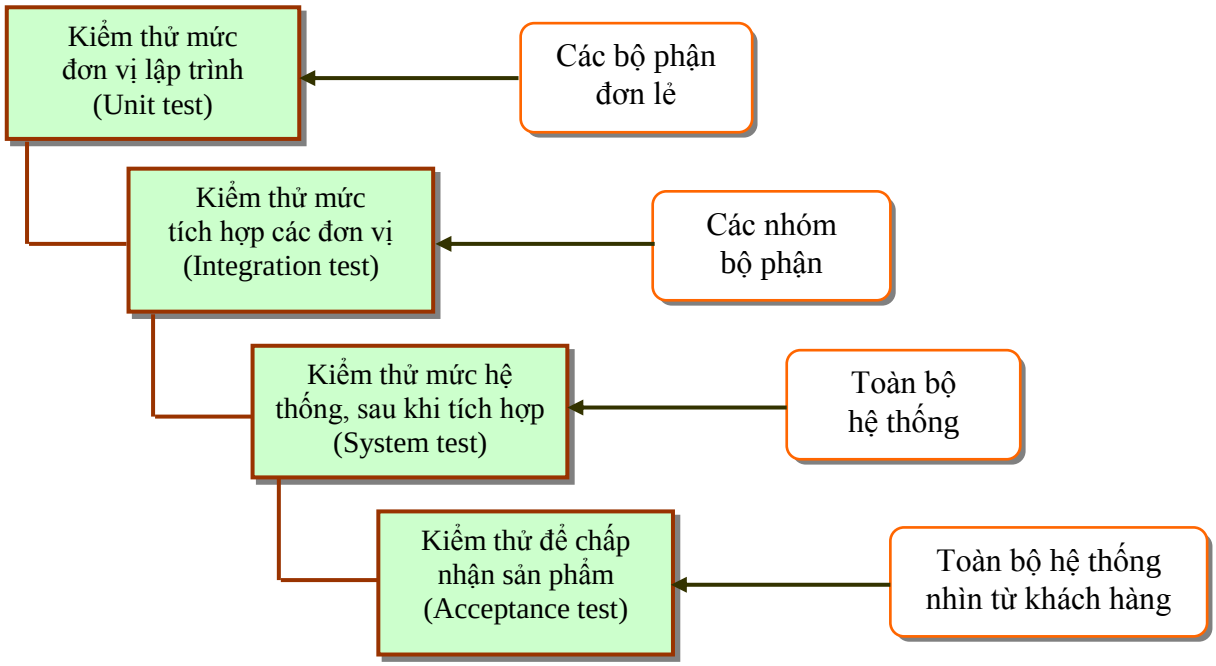
Kiểm thử phần mềm là quá trình thực thi một hệ thống phần mềm để xác định xem phần mềm có đúng với đặc tả không và thực hiện trong môi trường như mong đợi hay không.

Mục đích của kiểm thử phần mềm là tìm ra lỗi chưa được phát hiện, tìm một cách sớm nhất và bảo đảm rằng lỗi sẽ được sửa.

Mục tiêu của kiểm thử phần mềm là thiết kế tài liệu kiểm thử một cách có hệ thống và thực hiện nó sao cho có hiệu quả, nhưng tiết kiệm được thời gian, công sức và chi phí.

1.2. Các cấp độ kiểm thử phần mềm

Cấp độ kiểm thử phần mềm được thể hiện ở hình 1.1 [25]:



Hình 1.1- Bốn cấp độ cơ bản của kiểm thử phần mềm

1.2.1. Kiểm thử đơn vị (Unit Test)

Một đơn vị (Unit) là một thành phần phần mềm nhỏ nhất mà ta có thể kiểm thử được, ví dụ: các hàm (Function), thủ tục (Procedure), lớp (Class), hoặc các phương thức (Method).

1.2.2. Kiểm thử tích hợp (Integration Test)

Kiểm thử tích hợp kết hợp các thành phần của một ứng dụng và kiểm thử như một ứng dụng đã hoàn thành. Trong khi kiểm thử đơn vị kiểm tra các thành phần và Unit riêng lẻ thì kiểm thử tích hợp kết hợp chúng lại với nhau và kiểm tra sự giao tiếp giữa chúng.

1.2.3. Kiểm thử hệ thống (System Test)

Mục đích của kiểm thử hệ thống là kiểm thử xem thiết kế và toàn bộ hệ thống (sau khi tích hợp) có thỏa mãn yêu cầu đặt ra hay không.

Kiểm thử hệ thống kiểm tra cả các hành vi chức năng của phần mềm lẫn các yêu cầu về chất lượng như độ tin cậy, tính tiện lợi khi sử dụng, hiệu năng và bảo mật.

Kiểm thử hệ thống bắt đầu khi tất cả các bộ phận của phần mềm đã được tích hợp thành công.

Điểm khác nhau then chốt giữa kiểm thử tích hợp và kiểm thử hệ thống là kiểm thử hệ thống chú trọng các hành vi và lỗi trên toàn hệ thống, còn kiểm thử tích hợp chú trọng sự giao tiếp giữa các đơn thể hoặc đối tượng khi chúng làm việc cùng nhau. Thông thường ta phải thực hiện kiểm thử đơn vị và kiểm thử tích hợp để bảo đảm mọi Unit và sự tương tác giữa chúng hoạt động chính xác trước khi thực hiện kiểm thử hệ thống.

1.2.4. Kiểm thử chấp nhận sản phẩm (Acceptance Test)

Mục đích của kiểm thử chấp nhận là kiểm thử khả năng chấp nhận cuối cùng để chắc chắn rằng sản phẩm là phù hợp và thỏa mãn các yêu cầu của khách hàng và khách hàng chấp nhận sản phẩm.

Trong giai đoạn kiểm thử chấp nhận thì người kiểm tra là khách hàng. Khách hàng sẽ đánh giá phần mềm với mong đợi theo những thao tác sử dụng quen thuộc của họ. Việc kiểm tra ở giai đoạn này có ý nghĩa hết sức quan trọng tránh cho việc hiểu sai yêu cầu cũng như sự mong đợi của khách hàng.

1.3. Kỹ thuật kiểm thử phần mềm

Mục tiêu của kiểm thử là phải thiết kế các trường hợp kiểm thử có khả năng cao nhất trong việc phát hiện nhiều lỗi với thời gian và công sức tối thiểu. Do đó có thể chia các kỹ thuật kiểm thử thành hai loại:

- Kỹ thuật kiểm thử hộp đen (Black – box Testing) hay còn gọi là kỹ thuật kiểm thử chức năng (Functional Testing).
- Kỹ thuật kiểm thử hộp trắng (White – box Testing) hay còn gọi là kỹ thuật kiểm thử cấu trúc (Structural Testing).

1.3.1. Kỹ thuật kiểm thử hộp đen (Black – box Testing)

Kiểm thử hộp đen còn được gọi là kiểm thử hướng dữ liệu (data - driven) hay là kiểm thử hướng vào/ra (input/output driven).

Trong kỹ thuật này, người kiểm thử xem phần mềm như là một hộp đen. Người kiểm thử hoàn toàn không quan tâm đến cấu trúc và hành vi bên trong của chương trình. Người kiểm thử chỉ cần quan tâm đến việc tìm các hiện tượng mà phần mềm không hành xử theo đúng đặc tả của nó. Do đó, dữ liệu kiểm thử sẽ xuất phát từ đặc tả.

1.3.2. Kỹ thuật kiểm thử hộp trắng (White – box Testing)

Kiểm thử hộp trắng hay còn gọi là kiểm thử hướng logic, cho phép kiểm tra cấu trúc bên trong của phần mềm với mục đích bảo đảm rằng tất cả các câu lệnh và điều kiện sẽ được thực hiện ít nhất một lần. Người kiểm thử truy nhập vào mã nguồn chương trình và có thể kiểm tra nó, lấy đó làm cơ sở để hỗ trợ việc kiểm thử.

1.3. Kết luận

Trong chương 1 đã nêu tổng quan về các cấp độ và loại kiểm thử phần mềm cơ bản. Kiểm thử hộp trắng xem xét chương trình ở mức độ chi tiết và phù hợp khi kiểm tra các môđun nhỏ. Tuy nhiên, kiểm thử hộp trắng có thể không đầy đủ vì kiểm thử hết các lệnh không chứng tỏ là chúng ta đã kiểm thử hết các trường hợp có thể. Ngoài ra chúng ta không thể kiểm thử hết các đường đi đối với các vòng lặp lớn.

Kiểm thử hộp đen chú trọng vào việc kiểm tra các quan hệ vào ra và những chức năng giao diện bên ngoài , nó thích hợp hơn cho các hệ t hống phần mềm lớn hay các thành phần quan trọng của chúng . Nhưng chỉ sử dụng kiểm thử hộp đen là chưa đủ. Bởi vì, kiểm thử chức năng chỉ dựa trên đặc tả của môđun nên không thể kiểm thử được các trường hợp không được khai báo trong đặc tả. Ngoài ra, do không phân tích mã nguồn nên không thể biết được môđun nào của chương trình đã hay chưa được kiểm thử, khi đó phải kiểm thử lại hay bỏ qua những lỗi tiềm ẩn trong gói phần mềm.

Phương pháp kiểm thử hộp trắng và kiểm thử hộp đen là hai phương pháp cơ bản có vai trò rất quan trọng trong quá trình phát triển phần mềm, nếu chúng ta biết kết hợp chúng để bổ sung khiếm khuyết lẫn nhau.

CHƯƠNG 2 – KỸ THUẬT KIỂM THỬ ĐỘT BIẾN

2.1. Một số khái niệm

2.1.1. Kiểm thử đột biến

Kiểm thử đột biến được đề xuất đầu tiên vào năm 1979 bởi DeMillo và đồng nghiệp [13]. Nó cung cấp một phương tiện để đánh giá và cải tiến chất lượng dữ liệu thử cho chương trình được kiểm thử (PUT).

Kiểm thử đột biến tập trung vào việc đánh giá khả năng phát hiện lỗi của dữ liệu dùng để kiểm thử. Kiểm thử đột biến được dùng kết hợp với các kỹ thuật kiểm thử thông thường nhưng không thể được dùng để thay thế cho các kỹ thuật kiểm thử thông thường đó.

2.1.2. Đột biến

Kiểm thử đột biến bao gồm việc tạo ra các phiên bản lỗi của chương trình gốc được kiểm thử nhờ vào các toán tử đột biến. Các phiên bản lỗi đó được gọi là các *đột biến* (mutant).

Các *đột biến tương đương* (equivalent mutant) là các đột biến của chương trình gốc nhưng hoạt động hoàn toàn giống với chương trình gốc và cho ra kết quả giống với chương trình gốc trong mọi trường hợp kiểm thử.

2.1.3. Toán tử đột biến

Toán tử đột biến (mutation operator) là một luật được áp dụng vào chương trình gốc để tạo ra các đột biến. Các toán tử đột biến được xác định bởi ngôn ngữ của chương trình được kiểm thử và hệ thống đột biến được dùng để kiểm thử.

2.2. Cơ sở của kiểm thử đột biến

Kiểm thử đột biến là một kỹ thuật kiểm thử hộp trắng hay kiểm thử cấu trúc, được xây dựng dựa vào hai giả thuyết cơ bản [13]:

- Giả thuyết "lập trình viên giỏi" (competent programmer)
- Giả thuyết "hiệu ứng liên kết" (coupling effect).

2.3. Quy trình kiểm thử đột biến

Kiểm thử đột biến là một quy trình được lặp đi lặp lại để cải tiến dữ liệu thử đối với một chương trình và được chia thành các bước cơ bản [13] sau:

Bước 1: Sản sinh đột biến (dùng công cụ sản sinh tự động hoặc sản sinh thủ công) từ chương trình gốc.

Bước 2: Sản sinh các dữ liệu kiểm thử.

Bước 3: Thực hiện từng dữ liệu kiểm thử với chương trình gốc.

Bước 3.1: Nếu kết quả không đúng, phải chỉnh sửa lại chương trình và kiểm thử lại.

Bước 3.2: Nếu kết quả đúng, thực hiện bước tiếp theo.

Bước 4: Thực hiện từng dữ liệu kiểm thử với từng đột biến còn sống.

Bước 4.1: Nếu kết quả ra của đột biến khác với chương trình gốc, chương trình đột biến được xem là không đúng và *bị diệt*. Hoàn thành kiểm thử.

Bước 4.2: Nếu đột biến *sống sót* được (qua kiểm thử): phân tích các đột biến còn sống. Có hai khả năng xảy ra:

- Hoặc các đột biến là đột biến tương đương: không thể bị diệt.
- Hoặc có thể diệt các đột biến được nhưng các dữ liệu kiểm thử không đủ mạnh để diệt đột biến. Do đó phải tạo ra các dữ liệu kiểm thử khác và lặp lại bước 1.

2.4. Hạn chế của kiểm thử đột biến

Chi phí tính toán – tốn rất nhiều thời gian và công sức để thực hiện kiểm thử đột biến, và *tự động hóa* – để giảm công sức của kiểm thử viên.

2.5. Kết luận

Kiểm thử đột biến được giới thiệu để cung cấp một phương tiện để đánh giá và cải tiến chất lượng các bộ dữ liệu thử. Nó được xây dựng dựa trên hai giả thuyết cơ bản: giả thuyết lập trình viên giỏi, hiệu ứng liên kết. Do đó, kiểm thử đột biến chỉ tập trung vào các lỗi đơn giản của chương trình (ví dụ: sự khác biệt một từ đơn hoặc thay thế tên biến sai).

Tuy nhiên, chi phí để thực hiện kiểm thử đột biến khá cao vì một số lượng lớn các chương trình đột biến cần phải được thực hiện bởi ít nhất một dữ liệu thử và khó khăn để tự động hóa vì các dữ liệu thử mạnh cần phải được tạo ra, đột biến tương đương cần được loại bỏ, và kết quả đầu ra của PUT cần được kiểm thử tính đúng đắn. Vì vậy, chương 3 sẽ đề cập một số phương pháp cải tiến kỹ thuật kiểm thử đột biến để khắc phục các vấn đề trên.

CHƯƠNG 3 - MỘT SỐ CẢI TIẾN KỸ THUẬT KIỂM THỬ ĐỘT BIẾN

3.1. Giảm chi phí tính toán

Sử dụng phương pháp: *làm ít hơn*, *làm nhanh hơn* mà không làm giảm khả năng phát hiện lỗi.

3.1.1. Phương pháp làm ít hơn (A "do fewer" approach)

3.1.1.1. Lấy mẫu đột biến (Mutant Sampling)

Lấy mẫu đột biến là một phương pháp đơn giản lựa chọn ngẫu nhiên một tập con nhỏ các đột biến từ tập toàn bộ các đột biến.

Các nghiên cứu của Acree và Budd đề nghị rằng tỷ lệ lấy mẫu 10% có thể xác định trên 99% tất cả đột biến không tương đương trong khi cung cấp tiết kiệm chi phí đáng kể.

3.1.1.2. Đột biến ràng buộc (Constrained Mutation)

Giảm số lượng các đột biến cũng có thể đạt được bằng cách giảm số lượng các toán tử đột biến được áp dụng.

3.1.1.3. N - đột biến lựa chọn (N - Selective Mutation)

Sử dụng tất cả các toán tử đột biến trừ đi hai toán tử tạo ra hầu hết các đột biến (được gọi là phương pháp *2 - đột biến lựa chọn*). Giả thuyết phương pháp N – đột biến lựa chọn, trong đó N là số toán tử đột biến tạo ra nhiều đột biến nhất được loại bỏ. Ban đầu, 28 chương trình đã được kiểm tra để xác định tỷ lệ đột biến được tạo ra bởi mỗi toán tử đột biến, như thể hiện trong hình 3.2. Dựa trên đó, họ đề xuất loại bỏ hai toán tử SVR và ASR cho 2 - đột biến lựa chọn, cùng với SCR và CSR cho 4 - đột biến lựa chọn, kết hợp với ACR và SRC cho 6 - đột biến lựa chọn.

3.1.2. Phương pháp làm nhanh hơn (A “do smarter” approach)

3.1.2.1. Phương pháp lược đồ đột biến

Phương pháp tạo lược đồ đột biến (MSG) [18] là một trong những phương pháp giúp khắc phục vấn đề tắc nghẽn. Tất cả các đột biến của chương trình được mã hóa vào một mức mã nguồn duy nhất gọi là chương trình *siêu đột biến* (metamutant). Các điểm đột biến trong PUT (chẳng hạn như một phép toán số học) được thay thế bằng các lời gọi hàm có cú pháp hợp lệ được gọi là *siêu thủ tục* (metaprocedure). Mỗi metaprocedure mã hóa toán tử đột biến và thay đổi đầu ra của nó tùy thuộc vào các đối số.

3.1.2.2. Đột biến yếu (Weak Mutation)

Đột biến yếu khác với đột biến mạnh khi so sánh các trạng thái của đột biến và PUT. Cả hai phương pháp có thể được phân loại khi so sánh ở các thái cực đối lập: đột biến yếu so sánh ngay sau câu lệnh đột biến, còn đột biến mạnh so sánh khi kết thúc chương trình.

3.2. Tăng tự động hóa

3.2.1. Tạo dữ liệu thử tự động

Phát triển kỹ thuật *kiểm thử dựa trên ràng buộc* (CBT) để tạo ra các tập dữ liệu thử chất lượng dựa trên đột biến. Kỹ thuật này dựa trên khái niệm để diệt được đột biến thì dữ liệu thử phải thỏa mãn ba điều kiện:

1. *Điều kiện có thể đạt đến được*: Câu lệnh bị đột biến phải được kích hoạt).
2. *Điều kiện cần*: Khi đạt đến được câu lệnh bị đột biến, điều kiện *cần* là câu lệnh đột biến phải gây ra một trạng thái chương trình bên trong không đúng ngay sau khi nó được thực hiện.
3. *Điều kiện đủ*: Cuối cùng, dữ liệu thử là *đủ* nếu như trạng thái không đúng truyền qua đột biến dẫn đến thất bại khi kết thúc.

3.2.2. Xác định các đột biến tương đương tự động

Thực hiện các thuật toán dựa trên luồng dữ liệu và các kỹ thuật tối ưu hóa trình biên dịch để xác định các chương trình tương đương một cách tự động. Sử dụng sáu kỹ thuật tối ưu hóa trình biên dịch để xác định các đột biến tương đương được mô tả chi tiết hơn trong [13]:

- Xác định các đoạn mã chương trình chết (dead code) - Hình thức rõ ràng nhất: các đoạn mã lệnh không thể thực hiện được có chứa câu lệnh được sửa đổi sẽ là tương đương với PUT. Xác định thông qua đồ thị luồng điều khiển.
- Truyền hằng số (constant propagation) - Một biến có giá trị được xác định bằng hằng số khi chạy sẽ tạo ra các đột biến tương đương trong một số trường hợp. Ví dụ, hãy xem xét toán tử đột biến ABS. Một câu lệnh đột biến thực hiện giá trị tuyệt đối của một hằng số (≥ 0) sẽ luôn luôn tạo ra một đột biến tương đương.
- Truyền bất biến (invariant propagation) - Bất biến là một quan hệ đúng giữa hai biến hoặc một biến và một hằng số trong đó được xác định tại một điểm cụ thể trong chương trình. Ví dụ, để diệt một đột biến thay thế một biến bằng một biến khác, hai biến phải có giá trị khác nhau. Nếu biết được chúng đều bằng nhau tại điểm này, thì đột biến có khả năng là tương đương.
- Xác định biểu thức con chung - Sử dụng kết hợp với kỹ thuật truyền bất biến, xác định các biến tương đương dựa trên các biểu thức con chung. Ví dụ, nếu A = B + C - D và X = B + C - D thì A == X sau khi gán X. Thông tin này được sử dụng bằng kỹ thuật truyền bất biến để xác định các đột biến tương đương.
- Xác định bất biến vòng lặp - Tối ưu hóa code thường di chuyển mã bất biến ra bên ngoài vòng lặp. Nếu đột biến thay đổi vị trí này (tức là di chuyển mã vào bên trong hay ra bên ngoài vòng lặp) thì nó là tương đương.

– Nâng lên và hạ xuống (hoisting and sinking) - Tối ưu hóa chương trình cố gắng di chuyển đoạn mã chương trình được thực hiện nhiều lần đến vị trí mà nó được thực thi một lần. Các đột biến có ảnh hưởng bởi vị trí này là tương đương.

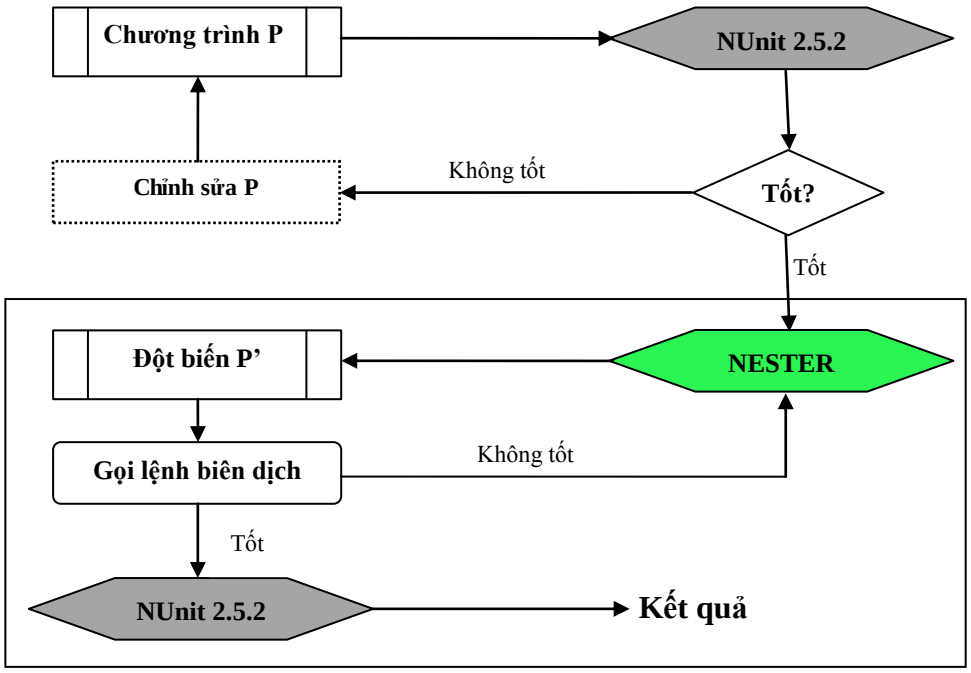
3.3. Kết luận

Kiểm thử đột biến được được xem là một kỹ thuật kiểm thử đơn vị mạnh để tìm ra tập dữ liệu thử hiệu quả giúp phát hiện các lỗi trong chương trình. Tuy nhiên, kiểm thử đột biến còn có một số hạn chế về chi phí tính toán và tự động hóa.

Offutt đã phát triển phương pháp kiểm thử dựa trên ràng buộc (CBT) để sản sinh dữ liệu thử tự động, các kết quả thực nghiệm là khá triển vọng, với CBT diệt được trung bình 97% các đột biến. Các kỹ thuật dựa trên tối ưu hoá trình biên dịch đã được áp dụng để phát hiện đột biến tương đương, các kết quả trong nghiên cứu cho thấy các phương pháp này phát hiện tỷ lệ trung bình 10% các đột biến tương đương.

CHƯƠNG 4 - ỨNG DỤNG KỸ THUẬT KIỂM THỬ ĐỘT BIẾN ĐỂ KIỂM THỬ CÁC CHƯƠNG TRÌNH C (C – SHARP)

Quy trình ứng dụng kiểm thử đột biến để kiểm thử các chương trình C-Sharp áp dụng kỹ thuật kiểm thử đột biến lựa chọn và sử dụng công cụ Nester, dùng để phân tích và tạo đột biến, và công cụ NUnit dùng để kiểm thử đơn vị. Quy trình này được minh họa trong Hình 4.1.



Hình 4.1. Quy trình ứng dụng kỹ thuật kiểm thử đột biến trong C-Sharp

4.1. Tìm hiểu về NUnit

NUnit là một framewwork miễn phí được sử dụng khá rộng rãi trong Unit Testing đối với ngôn ngữ .Net. Ban đầu được chuyển từ JUnit, phiên bản sản xuất hiện nay là phiên bản 2.5.

NUnit được viết hoàn toàn bằng C# và đã được hoàn toàn thiết kế lại để tận dụng lợi thế của nhiều người.

Ngôn ngữ .Net cho các thuộc tính tùy chỉnh các tính năng ví dụ và liên quan phản ánh khả năng khác.

4.2. Công cụ Nester

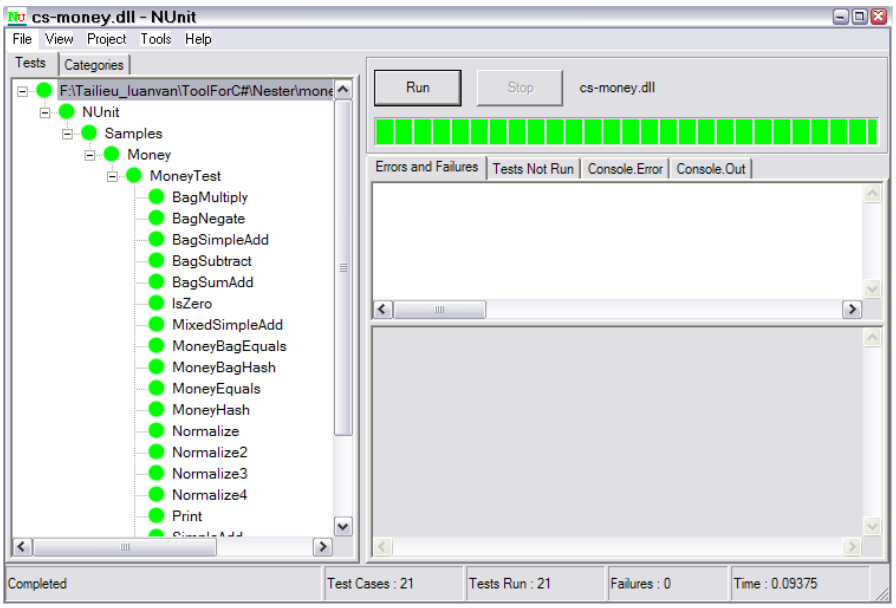
Nester là một công cụ miễn phí là hệ thống đột biến cho C-Sharp hỗ trợ toàn bộ quá trình đột biến cho các chương trình C-Sharp. Nó sản sinh các đột biến một cách tự động, thực thi các đột biến với các dữ liệu thử và báo cáo tỷ lệ đột biến của dữ liệu thử đó. Nó liên quan đến việc sửa đổi bổ sung các chương trình để có thể phân biệt các chương trình ban đầu từ các chương trình bị sửa đổi.

Phiên bản hiện tại của Nester là 0.3 Alpha hỗ trợ cho chương trình C-Sharp trong Microsoft Visual Studio 2005 và NUnit Framework.

4.3. Quy trình ứng dụng kiểm thử đột biến để kiểm thử các chương trình C - Sharp

4.3.1. Kiểm thử

Kiểm thử chương trình này bằng bộ kiểm thử NUnit (phiên bản 2.5.2), với các trường hợp kiểm thử và dữ liệu thử đã được thiết kế sẵn.



Dưới "góc nhìn" của NUnit với dữ liệu thử được xây dựng trong 21 trường hợp kiểm thử thì đây là một chương trình tốt .

Kết quả chạy NUnit:

Tên	Id	<1>	<2>	<3>	<4>	<5>	<6>	<7>	<8>	<9>	<10>	<11>	<12>	<13>	<14>	<15>	<16>	<17>	<18>	<19>	<20>	<21>	
NUnit.Samples.Money.MoneyTest.BagMultiply	<1>	6	21	20	20	20	13	21	14	5	6	5	16	7	14	13	5	5	20	16	5	5	41
NUnit.Samples.Money.MoneyTest.BagNegate	<2>	21	0	20	20	20	10	21	13	5	6	5	16	7	14	13	5	5	20	5	5	5	21
NUnit.Samples.Money.MoneyTest.BagSimpleAdd	<3>	20	20	0	43	43	29	22	14	5	8	6	35	25	35	33	5	21	43	8	8	21	43
NUnit.Samples.Money.MoneyTest.BagSubtract	<4>	20	20	43	0	43	29	22	14	5	8	6	35	25	35	33	5	21	43	8	8	21	43
NUnit.Samples.Money.MoneyTest.BagSumAdd	<5>	20	20	43	43	0	29	22	14	5	8	6	35	25	35	33	5	21	43	8	8	21	43
NUnit.Samples.Money.MoneyTest.IsZero	<6>	13	10	29	29	29	0	12	8	5	9	6	25	30	34	33	5	20	29	8	8	20	39
NUnit.Samples.Money.MoneyTest.MixedSimpleAdd	<7>	21	21	22	22	22	12	0	13	5	6	5	18	9	16	15	5	7	22	5	5	7	24
NUnit.Samples.Money.MoneyTest.MoneyBagEquals	<8>	14	13	14	14	14	8	13	2	5	8	8	14	7	7	7	5	6	14	6	6	6	21
NUnit.Samples.Money.MoneyTest.MoneyBagHash	<9>	5	5	5	5	5	5	5	5	0	5	5	5	5	5	5	5	5	5	5	5	5	5
NUnit.Samples.Money.MoneyTest.MoneyEquals	<10>	6	6	8	8	8	9	6	8	5	0	8	8	9	9	8	5	8	8	8	8	8	11
NUnit.Samples.Money.MoneyTest.MoneyHash	<11>	5	5	6	6	6	6	5	8	5	8	0	6	6	6	6	5	6	6	6	6	6	8
NUnit.Samples.Money.MoneyTest.Normalize	<12>	16	16	35	35	35	25	18	14	5	8	6	0	22	28	26	5	21	35	8	8	21	35
NUnit.Samples.Money.MoneyTest.Normalize2	<13>	7	7	25	25	25	30	9	7	5	9	6	22	0	35	34	5	19	25	8	8	19	35
NUnit.Samples.Money.MoneyTest.Normalize3	<14>	14	14	35	35	35	34	16	7	5	9	6	28	35	0	42	5	21	35	8	8	21	45
NUnit.Samples.Money.MoneyTest.Normalize4	<15>	13	13	33	33	33	33	15	7	5	8	6	26	34	42	0	5	20	33	8	8	20	42
NUnit.Samples.Money.MoneyTest.Print	<16>	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	23	5	5	5	5	5	28
NUnit.Samples.Money.MoneyTest.SimpleAdd	<17>	5	5	21	21	21	20	7	6	5	8	6	21	19	21	20	5	0	21	8	8	21	21
NUnit.Samples.Money.MoneyTest.SimpleBagAdd	<18>	20	20	43	43	43	29	22	14	5	8	6	35	25	35	33	5	21	0	8	8	21	43
NUnit.Samples.Money.MoneyTest.SimpleMultiply	<19>	16	5	8	8	8	8	5	6	5	8	6	8	8	8	8	5	8	8	0	8	8	19
NUnit.Samples.Money.MoneyTest.SimpleNegate	<20>	5	5	8	8	8	8	5	6	5	8	6	8	8	8	8	5	8	8	8	0	8	8
NUnit.Samples.Money.MoneyTest.SimpleSubtract	<21>	5	5	21	21	21	20	7	6	5	8	6	21	19	21	20	5	21	21	8	8	0	21
		41	21	43	43	43	39	24	21	5	11	8	35	35	45	42	28	21	43	19	8	21	Tổng số

Bảng 4.2. Kết quả chạy NUnit

Phân tích kết quả sau khi chạy NUnit :

1. Kiểm tra tất cả các mục, có giá trị như nhau trong một và tổng số các cột. Nhìn vào bảng kết quả chúng ta có thể thấy mẫu *BagNegative ()* <2> diệt tổng cộng 21 đột biến, và các đột biến tương tự cũng bị diệt *bởi MixedSimpleAdd* () <7 >. Điều đó không có nghĩa là chúng ta nên loại bỏ *BagNegative (),* bởi vì *MixedSimpleAdd ()* thay thế nó.

2. Kiểm tra tất cả các mục, có điểm đặc biệt số đột biến bị diệt thấp. Từ đó để chúng ta viết trường hợp thử nghiệm cho hợp lý. Rõ ràng là các vấn đề với việc kiểm tra quá nhỏ sẽ được tăng kích thước của mã kiểm tra đơn vị và bảo trì do đó tốn kém hơn.

3. Kiểm tra tất cả các mục, có tổng số các đột biến bị diệt cao.

4.3.2. Tạo đột biến

Sử dụng Nester với tập toán tử đột biến được lựa chọn để thực hiện đột biến. Tập toán tử đột biến được lựa chọn gồm: {true, false}, {+,-}, {==,!=}, {if(,if(true}, {xyz, a b c, 12, <> ?}, {1, 2, 3, 4}, {a, b, c, d}. Đây là các toán tử có xuất hiện trong chương trình và có thể bị "viết nhầm" bởi các lập trình viên.

Trong quá trình thực thi, Nester sinh ra số các đột biến là 78 và trong đó có 70 đột biến bị diệt và 8 đột biến "còn sống". Tỷ lệ đột biến ở đây là xấp xỉ 90%. Cụ thể với từng trường hợp kiểm thử được cho trong Bảng 4.3.

Bảng 4.3 – Chất lượng các trường hợp kiểm thử chương trình cs-money sau khi thực thi Nester.

Trường hợp kiểm thử	Số đột biến diệt được	Số đột biến không diệt được
1. BagMultiply	67	11
2. BagNegate	71	7
3. BagSimpleAdd	68	10
4. BagSubtract	67	11

5. BagSumAdd	68	10
6. IsZero	68	10
7. MixedSimpleAdd	71	7
8. MoneyBagEquals	71	7
9. MoneyBagHash	76	2
10. MoneyEquals	73	5
11. MoneyHash	76	2
12. Normalize	71	7
13. Normalize2	68	10
14. Normalize3	66	12
15. Normalize4	67	11
16. Print	76	2
17. SimpleAdd	74	4
18. SimpleBagAdd	68	10
19. SimpleMultiply	75	3
20. SimpleNegate	74	4
21. SimpleSubtract	73	5

Điều này chứng tỏ, chất lượng bộ dữ liệu thử được tạo ra trong 21 trường hợp kiểm thử ở trên rõ ràng là chưa cao, vì không có bất kỳ một trường hợp kiểm thử nào diệt được tất cả các đột biến. Đặc biệt, 4 đột biến được sinh ra khi Nester "chèn lỗi" vào lớp AssemlyInfo.cs, thì không có bất cứ trường hợp kiểm thử nào diệt được. Như vậy, Nester đã đưa ra được một cảnh báo rất kịp thời để chúng ta xem xét và xây dựng lại các trường hợp kiểm thử và bộ dữ liệu thử tốt hơn để đảm bảo chất lượng của phần mềm.

4.4. Kết luận

Kiểm thử đột biến được giới thiệu như là một ý tưởng để đánh giá chất lượng của các bộ dữ liệu kiểm thử. Dựa vào các ưu điểm, nhược điểm của kỹ thuật kiểm thử đột biến, có các phương pháp nhằm cải tiến kỹ thuật kiểm thử đột biến như ở chương 3; do đó ở chương 4 đã ứng

dụng để kiểm thử chương trình C-Sharp hiệu quả, giảm chi phí và thời gian.

Tuy nhiên từ chất lượng các trường hợp kiểm thử chương trình sau khi chạy Nester cho thấy chất lượng bộ dữ liệu thử được tạo ra trong 21 trường hợp kiểm thử ở trên rõ ràng là chưa cao, vì không có bất kỳ một trường hợp kiểm thử nào diệt được tất cả các đột biến. Vì vậy chúng ta cần xây dựng lại các trường hợp kiểm thử và bộ dữ liệu thử tốt hơn nhằm nâng cao chất lượng của chương trình cs – money.

KẾT LUẬN

Với cách tiếp cận dựa trên những đề xuất đã có trong lĩnh vực nghiên cứu về kiểm thử phần mềm, bản luận văn này là một sự tổng hợp những nét chính trong kiểm thử phần mềm nói chung và *kiểm thử đột biến* nói riêng cùng với những cải tiến. Sau đây là những điểm chính mà luận văn đã tập trung nghiên cứu:

- Trình bày một cách tổng quan nhất về kiểm thử phần mềm: khái niệm, mục đích và mục tiêu của kiểm thử, trong đó giới thiệu hai phương pháp thiết kế dữ liệu thử phổ biến được hầu hết các kiểm thử viên sử dụng hiện nay là phương pháp kiểm thử hộp trắng và phương pháp kiểm thử hộp đen, kèm theo các ví dụ.
- Giới thiệu kỹ thuật kiểm thử đột biến, các quy tắc để tạo đột biến và quy trình phân tích đột biến; các vấn đề còn hạn chế đối với kiểm thử đột biến, từ đó giới thiệu một số kỹ thuật cải tiến nhằm khắc phục những hạn chế trên.
- Sử dụng hai công cụ mã nguồn mở miễn phí Nester để tạo - phân tích đột biến, và NUnit để kiểm thử đơn vị và ứng dụng kiểm thử đột biến đối với chương trình C#; cụ thể là sử dụng kỹ thuật đột biến lựa chọn để kiểm thử chương trình cs – money với 21 trường hợp kiểm đạt tỷ lệ xấp xỉ 90%.

Trong quá trình thực hiện luận văn cũng như trong thời gian trước đó, tôi đã cố gắng tập trung nghiên cứu kỹ thuật kiểm thử này cũng như đã tham khảo khá nhiều tài liệu liên quan. Tuy nhiên, do thời gian và trình độ có hạn nên không tránh khỏi những hạn chế và thiếu sót nhất định. Tôi thật sự mong muốn nhận được những góp ý cả về chuyên môn lẫn cách trình bày của luận văn từ bạn đọc.

HƯỚNG PHÁT TRIỂN

Kiểm thử đột biến là một kỹ thuật kiểm thử được khá nhiều nhà nghiên cứu quan tâm bởi tác dụng của nó. Tuy nhiên, trong luận văn này, vẫn còn tồn tại nhiều vấn đề, trong thời gian tới, tôi cần phải tiếp tục nghiên cứu:

- Các vấn đề về phát hiện đột biến tương đương trong chương trình được kiểm thử.
- Tìm hiểu thêm các phương pháp khác nhằm giảm chi phí tính toán và tăng tính tự động hoá cho chương trình được kiểm thử.
- Đối với ứng dụng kỹ thuật kiểm thử đột biến để kiểm thử chương trình cs - money, luận văn chỉ mới dừng lại ở mức độ đánh giá chất lượng 21 trường hợp kiểm thử đã được xây dựng ở trên, chưa cải tiến nó để đạt tỷ lệ đột biến 100%. Do đó, trong thời gian tới, tôi sẽ tiếp tục nghiên cứu để loại bỏ các đột biến tương đương trong số các đột biến còn sống của chương trình này, đồng thời cải tiến các trường hợp kiểm thử để đạt tỷ lệ đột biến 100%.

Cuối cùng, tôi hy vọng sau khi giải quyết xong những vấn đề còn tồn tại nêu trên, tôi sẽ tiếp tục nghiên cứu kiểm thử đột biến để ứng dụng cho các ngôn ngữ khác như: JAVA, SQL, ….Bởi vì, qua quá trình nghiên cứu kỹ thuật kiểm thử đột biến, tôi thấy có rất nhiều công cụ hỗ trợ để thực hiện kiểm thử đột biến cho các ngôn ngữ này.

References.

Tiếng Việt

[1] Nguyễn Xuân Huy (1994), *Công nghệ phần mềm*, Trường Đại học Tổng hợp TP.HCM.

[2] Lê Văn Tường Lân (2004), *Giáo trình công nghệ phần mềm*, Trường Đại học Khoa học Huế - Đại học Huế.

[3] Hồ Văn Phi (2008), N*ghiên cứu và ứng dụng kiểm thử đột biến cho các câu lệnh truy vấn SQL,* Luận văn thạc sỹ kỹ thuật, chuyên ngành khoa học máy tính, trường Đại học Bách Khoa - Đại học Đà Nẵng.

Tiếng Anh

[4] A.P. Mathur (1991), *Performance, effectiveness and reliability issues in software testing*, Tokyo, Japan.

[5] A.J.Offutt and K.N.King, "A Fortran 77 interpreter for mutation analysis", *in 1987 Symposium on Interpreters and Interpretive Techniqes*, pp. 177-188, ACM SIGPLAN, June 1987.

[6] A.J. Offutt and W.M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *The Journal of Softwave Testing, Verification, and Reliability*, vol.4, pp. 131-154, September 1994.

[7] A.T. Acree, T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Mutation Analysis", Georgia Institute of Technology, *Technical Report*, GIT – ICS – 79/08, 1979.

[8] A.T.Acree, *On Mutation*, PhD thesis, Georgia Institute of Technology, Atlanta GA, 1980.

[9] Jeff Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf (1996), *An Experimental Determination of Sufficient Mutant Operators*, George Mason University.

[10] Jeff Offutt, Gregg Rothermel, Roland H. Untch, and Christian Zapf (1993), *An Experimental Evaluation of Selective Mutation*, Baltimore, MD.

[11] K.S.H.T.Wah, "Fault coupling in finite bijecttive functions," *the Journal of Softwave testing Verification, and Reliability*, vol.5, pp.3-47, March 1995.

[12] Mark Harman and Rob Hierons (2006), "An Overview of Mutation Testing", *Genetic Algorithms for Mutation Testing,* Brunel University, London.

[13] R.A. DeMillo and A.J. Offutt (1993), *Experimental results from an automatic test case generator,* ACM transactions on Softwar Engineering Methodology, 2(2) pages 109-127.

[14] R.A. DeMillo, D.S. Guindi, K.N.King, W.M.Mc Cracken and A.J.Offutt, "An extended overview of the Mothra Softwave testing environment," *in Proceeding of the Second wrokshop on Softwave Testing, Verification, and Anlysis*, (Banff Alberta) pp. 142-151, IEEE Computer Society Press, July 1988.

[15] R.Geist, A.J.Offutt, and F. Harris, "Estimation and endhancement of real-time softwave reliability through mutation analysis," *IEEE Transactions on Computers*, vol.41, pp. 550-558, May 1992. Special Issue on Fault – Tolerant Computing.

[16] R.Untch (1992), "Mutation-based software testing using program schemata", *Proceedings of 30th ACM Southeast Regional Conference*, Raleigh, NC.

[17] R.J.Lipton and F.G. Sayward, "the status of reseach on program mutation," *in Digest for the Workshop on Softwave Testing and Test Documentation*, pp. 355-373, December 1978.

[18] R. Untch, A.J. Offutt and M.J. Harold (1993), *Mutation Analysis using program schemate*, pages 139-148, Cambridge, MA.

[19] T.A. Budd (1980), *Mutation Analysis of Program Test Data*, Ph.D Thesis, Yale University, New Haven CT.

[20] Nguyen Thanh Binh, C. Robach (2001), "Mutation Testing Applied

to Hardware: the Mutants Genenration", *Proceedings of the 11th IFIP International Conference on Very Large Scale Integration*,118--123, Montpellier, France.

[21] W.E. Howden (1982), Weak mutation testing and completeness of test sets, *IEEE Transactions on Software Engineering*, 8(4) pages 371-379.

[22] W.Wong, J.Maldonado, an M.Delamaro Peducing *the cost of regression test by using selective mutation*. In 8 th CITS – International Conference on Softwave Technology, pages 11 – 13, Curitiba, PR, June 1997.

[23] W.E.Wong, *On Mutation and Data Flow*. PhD thesis, Purduce University, December 1993. (Also Technical Re-port SERC – TR – 149 – P, Softwave Engineering Research Center, Purduce University, West Lafayetle.

[24] W.E.Wong, M.E. Delamaro, J.C. Maldonado, and A.p.Mathur, "Constrained mutation in C program" *in proceedings of the 8th Brarilian Symposium on Softwave Engineering*, (Curitiba, Brazil), pp. 439 – 452, October 1994.

[25] http://www.scribd.com