Kỷ yếu Hội nghị Quốc gia lần thứ X về Nghiên cứu cơ bản và ứng dụng Công nghệ thông tin (FAIR), Đà Nẵng, ngày 17-18/08/2017
DOI: 10.15625/vap.2017.00040

KIỂM THỬ ĐỘT BIẾN BẬC CAO: HIỆU QUẢ VÀ NHỮNG VẤN ĐỀ TỒN TẠI

Đỗ Văn Nhỏ[1], Nguyễn Quang Vũ[2], Nguyễn Thanh Bình[3]

[1]Trường THPT chuyên Lê Quý Đôn, Đà Nẵng

[2] Trường CĐ CNTT Hữu nghị Việt Hàn

[3]Trường Đại học Bách khoa, Đại học Đà Nẵng

dovannho@gmail.com, vunq@viethanit.edu.vn, ntbinh@dut.udn.vn

TÓM TẮT: *Kiểm thử đột biến (còn được gọi là kiểm thử đột biến bậc 1) được giới thiệu như là một kỹ thuật để đánh giá chất lượng của các ca kiểm thử. Đây là một kỹ thuật tự động và khả dụng cho việc đánh giá hiệu quả phát hiện lỗi của kiểm thử phần mềm. Tuy nhiên, có 3 thách thức chính trong việc ứng dụng kiểm thử đột biến: (1) số lượng đột biến quá lớn (điều này dẫn đến chi phí tính toán rất cao); (2) các đột biến được sinh ra có phản ánh lỗi thực hay không; (3) vấn đề đột biến tương đương. Có rất nhiều giải pháp đã được đề xuất để giải quyết các hạn chế này. Mỗi một giải pháp đều có những ưu và nhược điểm riêng. Trong số đó, kiểm thử đột biến bậc cao, được giới thiệu đầu tiên vào năm 2009, là một giải pháp có khả năng hạn chế đồng thời cả 3 vấn đề nêu trên và đặc biệt giải quyết vấn đề các lỗi được chèn vào phản ánh được lỗi thực. Trong bài báo này, chúng tôi đề cập đến hiệu quả cũng như sự phát triển của các giải pháp trong lĩnh vực kiểm thử đột biến bậc cao. Trên cơ sở đó, chúng tôi đánh giá những vấn đề tồn tại của các giải pháp và đề xuất những hướng nghiên cứu trong tương lai.*

Từ khóa: *Kiểm thử phần mềm; Kiểm thử đột biến; Kiểm thử đột biến bậc cao; Hạn chế của kiểm thử đột biến.*

I. GIỚI THIỆU

Quy trình phát triển phần mềm thường trải qua nhiều giai đoạn khác nhau. Trong đó, kiểm thử phần mềm là hoạt động quan trọng nhằm phát hiện lỗi và đảm bảo chất lượng của phần mềm. Thất bại trong việc phát hiện lỗi có thể dẫn đến những tổn thất kinh tế hoặc thậm chí là thảm hoạ trong trường hợp hệ thống cần sự hoạt động an toàn tuyệt đối. Tuy nhiên kiểm thử phần mềm là hoạt động rất tốn kém, nó chiếm khoảng từ 40%-50% tổng chi phí cho sự phát triển nói chung. Riêng các hệ thống đòi hỏi độ tin cậy, an toàn cao thì hoạt động này tiêu tốn chi phí nhiều hơn [1].

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?

Kiểm thử đột biến [1] là một kỹ thuật cho phép đánh giá chất lượng của dữ liệu thử, tức là khả năng phát hiện lỗi, bằng cách chèn lỗi vào chương trình cần kiểm thử. Sau đó, kiểm tra xem dữ liệu thử có phát hiện được lỗi chèn vào không. Từ khi ra đời, kiểm thử đột biến đã được ứng dụng rộng rãi cho nhiều ngôn ngữ lập trình cũng như nhiều lĩnh vực phần mềm khác nhau. Đồng thời, nhiều nghiên cứu được đề xuất để cải tiến và nâng cao chất lượng của ứng dụng kiểm thử đột biến. Trong đó, kiểm thử đột biến bậc cao được đề xuất gần đây [17] cho phép giải quyết các hạn chế cơ bản của kiểm thử đột biến truyền thống, như số lượng đột biến lớn, sự phản ánh lỗi thực của đột biến và đột biến tương đương. Vì vậy, trong bài báo này, chúng tôi tập trung nghiên cứu về kiểm thử đột biến bậc cao nhằm tiếp tục nâng cao hiệu quả và ứng dụng vào thực tế kiểm thử phần mềm.

Bài báo được tổ chức gồm sáu mục. Mục II sẽ trình bày các khái niệm cơ bản và phân tích các hạn chế của kiểm thử đột biến. Kiểm thử đột biến bậc cao được trình bày trong Mục III. Trong Mục IV, chúng tôi phân tích các nghiên cứu liên quan về kiểm thử đột biến bậc cao. Mục V trình bày các hạn chế của kiểm thử đột biến bậc cao, từ đó đề xuất các hướng nghiên cứu. Mục cuối cùng của bài báo là phần đánh giá kết luận.

II. KIỂM THỬ ĐỘT BIẾN

Dick Lipton là người đầu tiên đề xuất kỹ thuật kiểm thử đột biến [1], sau đó lĩnh vực này được đánh dấu sự ra đời và phổ biến bởi DeMillo [2] và Sayward [3]. Kiểm thử đột biến là kỹ thuật kiểm thử dựa trên lỗi nhằm cung cấp một tiêu chuẩn kiểm thử được gọi là tỷ lệ đột biến. Tỷ lệ đột biến được sử dụng để đánh giá khả năng phát hiện lỗi của tập dữ liệu thử.

Nguyên lý chung của kiểm thử đột biến là tạo ra các phiên bản của chương trình có chứa các lỗi đơn giản, các lỗi được sử dụng bởi kiểm thử đột biến đại diện cho các lỗi sơ suất do lập trình viên thường tạo ra. Các lỗi như thế được gieo một cách thận trọng vào chương trình gốc, bằng cách thay đổi cú pháp đơn giản, để tạo một tập các chương trình lỗi. Mỗi chương trình lỗi được gọi là đột biến, mỗi đột biến mang một thay đổi cú pháp khác nhau. Cụ thể, cho P là một chương trình gốc, P' là một đột biến của P bằng cách thực hiện một thay đổi nhỏ về cú pháp trong chương trình.

Chẳng hạn, trong Bảng 1, P là chương trình gốc, P' và P'' là các đột biến của P bằng cách thay đổi cú pháp trong phép toán quan hệ, thay thế $(x > y)$ bởi $(x < y)$ đối với P'; thay $(x > y)$ bởi $(x >= y)$ đối với P''. Các câu lệnh bị đột biến được đánh dấu bởi ký hiệu gạch chân.

Bảng 1. Ví dụ minh họa các đột biến

Chương trình gốc *P*	**Đột biến *P'***	**Đột biến *P''***

`int max(int x,int y)`
`{`
`  int mx=x;`
`  if (x < y)`
`  mx=y`
`  return mx;`
`}`
	`int max(int x,int y)`
	`{`
	`  int mx=x;`
	`  if (x > y)`
	`  mx=y`
	`  return mx;`
	`}`
		`int max(int x,int y)`
		`{`
		`  int mx=x;`
		`  if (x <= y)`
		`  mx=y`
		`  return mx;`
		`}`

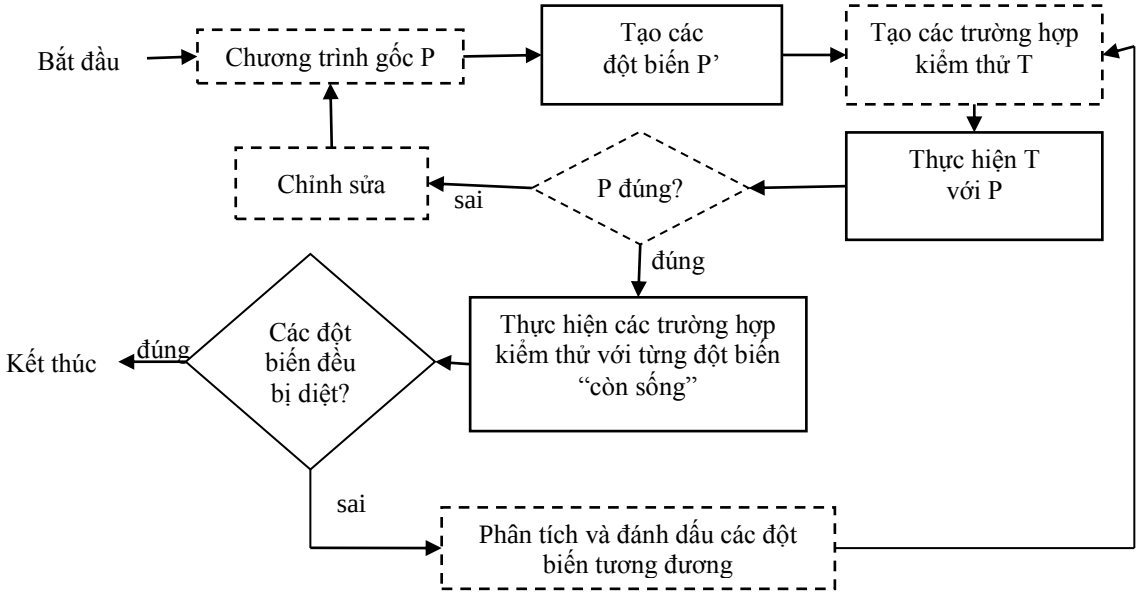
Sau khi tạo ra các đột biến, mỗi đột biến và chương trình gốc được thực thi trên cùng một bộ dữ liệu thử, nếu kết quả của đột biến và chương trình gốc khác nhau, thì đột biến đó được gọi là đột biến bị diệt. Nếu không thể tìm thấy dữ liệu thử sao cho khi thực thi đột biến và chương trình gốc cho kết quả khác nhau, thì đột biến đó được gọi là đột biến tương đương. Chẳng hạn, trong ví dụ trên, P'' là đột biến tương đương. Vì P và P'' luôn cho cùng kết quả với mọi dữ liệu thử. Sau khi thực thi trên tất cả các dữ liệu thử, tỷ lệ đột biến (Mutation Score - MS) được tính toán bằng tỷ lệ phần trăm của số các đột biến bị diệt chia cho số đột biến không tương đương. Mục tiêu của kiểm thử viên là tạo ra bộ dữ liệu thử có tỷ lệ đột biến bằng 1, nghĩa là tất cả các đột biến đều bị diệt. Khi đó, bộ dữ liệu thử phát hiện được tất cả các lỗi chèn vào trong chương trình.

Tỷ số $MS = 100 * D/(N - E)$ được gọi là tỷ lệ đột biến.

Trong đó, MS: tỷ lệ đột biến; D: là đột biến đã bị diệt; N: là tổng số các đột biến; E: là tổng số đột biến tương đương. Tỷ lệ đột biến cho phép đánh giá chất lượng bộ dữ liệu thử.

Kiểm thử đột biến hứa hẹn sẽ hiệu quả trong việc xác định dữ liệu thử thích hợp có thể được dùng để phát hiện các lỗi thực sự [4]. Tuy nhiên, số lượng các lỗi tiềm năng cho một chương trình nhất định là rất lớn, không thể tạo ra đột biến đại diện cho tất cả các lỗi. Vì vậy, kiểm thử đột biến truyền thống chỉ nhằm mục tiêu tạo một tập con của các phiên bản lỗi, là những phiên bản gần với phiên bản đúng của chương trình với hy vọng rằng đó sẽ là đủ để mô phỏng cho tất cả các lỗi. Lý thuyết này dựa trên hai giả thuyết cơ bản: giả thuyết "Lập trình viên giỏi" (Competent Programmer Hypothesis - CPH) [2-3] và giả thuyết "Hiệu ứng liên kết" (Coupling Effect Hypothesis - CEH) [2].

Giả thuyết lập trình viên giỏi được giới thiệu đầu tiên do DeMilo và các đồng nghiệp [2]. Giả thuyết này cho rằng các lập trình viên thông thường rất giỏi, chương trình họ viết ra nếu có sai sót thì đó là những sai sót rất nhỏ so với chương trình đúng. Các lỗi phạm phải bởi lập trình viên là các lỗi đơn giản. Do vậy, với kiểm thử đột biến, chỉ các lỗi đơn giản được tạo ra bởi việc thực hiện các thay đổi nhỏ về cú pháp được áp dụng.



Hình 1. Quy trình kiểm thử đột biến

Giả thuyết hiệu ứng liên kết cũng được đề xuất bởi DeMillo và các đồng nghiệp [2]. Không giống như các CPH liên quan đến hành vi của một lập trình viên, hiệu ứng liên kết liên quan đến các lỗi được sử dụng trong phân tích đột biến. Giả thuyết này cho rằng "các lỗi phức tạp được liên kết từ các lỗi đơn giản, như vậy bộ dữ liệu thử đủ khả năng

phát hiện tất cả các lỗi đơn giản trong chương trình thì cũng có khả năng phát hiện các lỗi phức tạp với tỉ lệ cao". Các công trình nghiên cứu lý thuyết cũng như thực nghiệm khẳng định giả thuyết hiệu ứng liên kết là đúng [7-9].

Kiểm thử đột biến là một quy trình được thực hiện lặp đi lặp lại qua nhiều hoạt động khác nhau để cải tiến dữ liệu thử đối với một chương trình. Hình 1 minh họa quy trình kiểm thử đột biến. Trong quy trình này, các bước biểu diễn bởi nét liền là có thể tự động hóa, các bước biểu diễn bằng nét đứt thường được xử lý thủ công.

Các tham số ban đầu để xử lý là chương trình gốc P (cần được kiểm thử) và bộ dữ liệu thử T. Trước tiên, bằng cách phán xét, chương trình gốc P phải được thực hiện với các dữ liệu thử trong T. Nếu có bất kỳ kết quả đầu ra nào không đúng thì T đã cho thấy rằng chương trình gốc P chứa lỗi. Các lỗi này phải được chỉnh sửa trước khi tiếp tục quy trình.

Khi đã xác định được tất cả các dữ liệu thử trong T cung cấp các đầu ra chính xác, bước tiếp theo là tạo ra một tập M - tập các đột biến P' của chương trình gốc P. Sau khi tạo ra tập M chứa tất cả các đột biến, lần lượt các đột biến P' này được thực hiện với T và các kết quả đầu ra của chúng được so sánh với các kết quả đầu ra của chương trình gốc P, lúc này có hai kịch bản khác nhau có thể xảy ra dưới đây.

- Lỗi được chèn vào trong chương trình đột biến P' được nhận biết, nghĩa là chương trình P và đột biến P' cho ra các kết quả khác nhau. Trong trường hợp này, đột biến P' được gọi là bị diệt (killed) bởi dữ liệu thử T. Khi đó, T được gọi là dữ liệu thử thích hợp vì nó có khả năng phát hiện được sự khác nhau giữa chương trình gốc P và đột biến P'. Các đột biến như vậy trở nên không cần thiết vì dữ liệu thử hiện có đã phân biệt được chúng và do đó chúng sẽ bị loại khỏi tập đột biến M.

- Chương trình gốc P và đột biến P' cho ra kết quả hoàn toàn giống nhau. Trong trường hợp này, có thể có hai khả năng xảy ra. Khả năng thứ nhất là dữ liệu thử T không đủ tốt (hay được gọi là dữ liệu thử không thích hợp), chúng ta sẽ phải tiến hành thực hiện kiểm thử lại với các dữ liệu thử tốt hơn. Khả năng thứ hai là chương trình P và đột biến P' là những chương trình tương đương nhau, mọi dữ liệu thử đều không thể phân biệt sự khác nhau giữa chúng do đó P' được gọi là đột biến tương đương. Trong cả hai trường hợp này, đột biến P' được cho là còn sống (alive).

Khi tất cả các dữ liệu thử trong T đã được thực hiện trên tất cả các đột biến trong M, các đột biến đó vẫn còn sống (vẫn còn tồn tại trong M) tức là cho đến lúc này vẫn không thể phân biệt chúng với chương trình gốc P. Nói cách khác, không tồn tại dữ liệu thử trong T làm cho những đột biến còn sống tính toán kết quả đầu ra khác so với P. Những đột biến này trở thành mục tiêu cho lần lặp tiếp theo, trong đó dữ liệu thử mới sẽ được tạo ra với nỗ lực để phân biệt được chúng.

Quá trình này tiếp tục cho đến khi tất cả các đột biến trong M bị diệt. Tuy nhiên, để diệt được các đột biến không phải là một công việc dễ dàng vì một số đột biến có thể có ngữ nghĩa giống với chương trình gốc. Những đột biến tương đương này sẽ luôn luôn tạo ra kết quả đầu ra giống với với bất kỳ dữ liệu vào nào.

Kỹ thuật kiểm thử đột biến truyền thống [8] (hay còn được gọi là kiểm thử đột biến bậc một) được đề cập ở trên như là một phương pháp có sự tự động hóa và hiệu quả cao trong việc đánh giá chất lượng của các bộ dữ liệu thử. Nó có thể được áp dụng cho kiểm thử phần mềm, với nhiều ngôn ngữ lập trình khác nhau và tại nhiều mức kiểm thử khác nhau, như kiểm thử đơn vị, kiểm thử tích hợp, kiểm thử hệ thống, kiểm thử chấp nhận,… Tuy nhiên, nó vẫn chưa được áp dụng rộng rãi vì vẫn còn tồn tại ba hạn chế chính [9] dưới đây:

- Số lượng các đột biến được sinh ra quá nhiều. Một chương trình có thể bao gồm rất nhiều toán tử ở các dòng lệnh khác nhau, vì vậy số lượng các đột biến được tạo ra là rất lớn. Ví dụ, có một chương trình đơn giản chỉ với một câu lệnh chính là trả về giá trị của phép toán "$a + b$", chúng ta có thể có các đột biến chứa các phép toán khác, như "$a - b$","$a * b$","a/b","$a + b + +$","$a + 0$","$0 + b$", ... Số lượng các đột biến lớn sẽ dẫn đến vấn đề chi phí tính toán lớn bởi vì những ca kiểm thử không chỉ được thực hiện trên chương trình gốc mà còn phải thực hiện trên tất cả các đột biến được sinh ra.

- Vấn đề liệu các đột biến đó có mô tả đúng các lỗi thực sự của phần mềm hay không. Các đột biến được sinh ra bởi việc chèn lỗi đơn giản (áp dụng các toán tử đột biến), do đó nó sẽ có thể không chứng tỏ một cách chính xác các lỗi thực của phần mềm, vì các lỗi này quá đơn giản. Theo Langdon và các cộng sự [20-21], 90% lỗi của các phần mềm là các lỗi phức tạp.

- Vấn đề các đột biến tương đương. Trong thực tế, có rất nhiều các toán tử đột biến có thể được dùng để tạo ra các đột biến tương đương có cùng "hành vi" với chương trình gốc. Trong trường hợp này, không thể có bất kỳ một dữ liệu thử nào phát hiện được sự khác nhau giữa chương trình gốc và đột biến tương đương của nó.

Trong số đó, kiểm thử đột biến bậc cao không chỉ là kỹ thuật duy nhất có thể cải tiến được hạn chế thứ hai mà còn là kỹ thuật duy nhất đồng thời có thể cải tiến cả ba hạn chế của kỹ thuật đột biến truyền thống [9-14]. Chi tiết về đột biến bậc cao sẽ được trình bày trong phần tiếp theo.

Bảng 2. Các đề xuất cải tiến ba hạn chế của kiểm thử đột biến bậc một

Giảm số lượng đột biến sinh ra	Tăng độ thực của lỗi	Giảm đột biến tương đương
- Đột biến bậc cao - Đột biến mẫu - Đột biến lựa chọn - Đột biến nhóm - Đột biến yếu - Tối ưu thời gian thực thi	- Đột biến bậc cao	- Đột biến bậc cao - Tối ưu hóa biên dịch - Ràng buộc toán học - Sử dụng mô hình kiểm tra Lesar - Đột biến lựa chọn - Đồng tiến hóa - Phân tích phụ thuộc chương trình - Điều kiện tương đương - Phân cấp lỗi - Phân cấp ngữ nghĩa ngoại lệ

III. KIỂM THỬ ĐỘT BIẾN BẬC CAO (Higher Order Mutation Testing - HOM)

Theo Harman và các cộng sự [17], các đột biến được phân thành hai loại: Đột biến bậc một (FOM- First Order Mutants) và Đột biến bậc cao (HOM - Higher Order Mutants). FOM được tạo bằng cách áp dụng toán tử đột biến một lần duy nhất và được sử dụng trong kiểm thử đột biến bậc một, trong khi đó HOM sử dụng nhiều hơn một đột biến và được sử dụng trong kiểm thử đột biến bậc cao.

Trong ví dụ sau (Bảng 3), từ chương trình gốc P, đột biến bậc một P' thay đổi dòng lệnh $r = x + 1$ bằng dòng lệnh $r = x - 1$ trong khi đột biến bậc cao (ở đây là bậc hai) thực hiện thay đổi ở hai dòng lệnh được đánh dấu * so với chương trình gốc.

Bảng 3. Ví dụ minh họa về đột biến bậc hai

Chương trình gốc *P*	Đột biến bậc nhất *P'*	Đột biến bậc cao (bậc hai) *P''*

`int foo(int x){`
`  int r=x+1;`
`  r=r-1;`
`  return r`
`}`
	`int foo(int x){`
	`  int r=x-1;*`
	`  r=r-1;`
	`  return r`
	`}`
		`int foo(int x){`
		`  int r=x-1;*`
		`  r=r+1; *`
		`  return r`
		`}`

Trước đây, nói đến kiểm thử đột biến chính là nói đến kiểm thử bậc một. Theo quan điểm này, kiểm thử đột biến là qui trình chèn một lỗi đơn vào chương trình gốc. Quan điểm này trở thành phổ biến và được chấp nhận rộng rãi trong cộng đồng nghiên cứu về kiểm thử bởi người ta tin rằng kiểm thử bậc cao quá tốn kém vì vậy không thực tế. Hơn nữa, nhiều nhận định cho rằng hiệu ứng liên kết làm cho kiểm thử bậc cao là không quan trọng bởi vì chúng là sự kết đôi của kiểm thử bậc một.

Tuy nhiên, những kết quả nghiên cứu thực nghiệm gần đây dùng đột biến bậc cao đã phản ánh tốt hơn các lỗi thực tế. Purushthaman và Perry [15] chỉ ra rằng 90% các lỗi sau khi vận hành và triển khai là lỗi phức tạp. Các lỗi phức tạp này có thể có được bằng việc thay đổi một số cú pháp của chương trình ở một vài vị trí. Vấn đề này được giải quyết đơn giản bằng đột biến bậc cao. Tương tự, Eldh và cộng sự [5] chỉ ra hơn 50% là lỗi phức tạp. Vì vậy, đột biến bậc cao thực sự đại diện cho các lỗi phức tạp và khó bị diệt.

Cụ thể, trong một ví dụ được chỉ ra bởi Yue Jia [16] khi thực hiện so sánh kiểm thử đột biến bậc một và bậc cao. Với chương trình gốc TCAS (một chương trình tránh va chạm của máy bay) gồm 20 dòng lệnh. Đột biến bậc một gồm FOM 1, FOM 2, FOM 4 được tạo bằng cách phủ định các biểu thức điều kiện, FOM 3 thay thế toán tử == bằng !=. Các đột biến bậc cao HOM 1 kết hợp FOM 1 và FOM 2, HOM 2 kết hợp FOM 3 và FOM 4. Trên tổng số 1608 ca kiểm thử được thực hiện, kết quả được trình bày trong Bảng 4.

Bảng 4. Ví dụ về kết quả của đột biến bậc cao

Đột biến	Số ca kiểm thử bị diệt	Nhận xét
FOM 1	886	
FOM 2	269	
HOM 1 (FOM 1+FOM 2)	125	HOM 1 khó bị diệt hơn FOM 1 và FOM 2 và bất kỳ ca kiểm thử bị diệt nào của FOM 1 và FOM 2 cũng đều bị diệt bởi HOM 1 nhưng không có điều ngược lại.
FOM 3	224	
FOM 4	260	
HOM 2 (FOM 3+FOM 4)	40	HOM 2 khó bị diệt hơn FOM 3 và FOM 4 và không một ca kiểm thử nào diệt FOM 3 và FOM 4 có thể diệt HOM 2.

Để làm cho kiểm thử bậc cao trở nên thực tế hơn, Harman và các cộng sự [17] đã áp dụng các kỹ thuật tìm kiếm tối ưu. Điều này cung cấp phương tiện để khám phá không gian của đột biến bậc cao một cách hiệu quả, được hướng dẫn bằng hàm thích nghi (fitness function) nhằm tìm ra đột biến có chất lượng. Trong nghiên cứu của mình [7], Harman và các cộng sự đã áp dụng các thuật toán tìm kiếm tối ưu hóa đơn mục tiêu và đa mục tiêu.

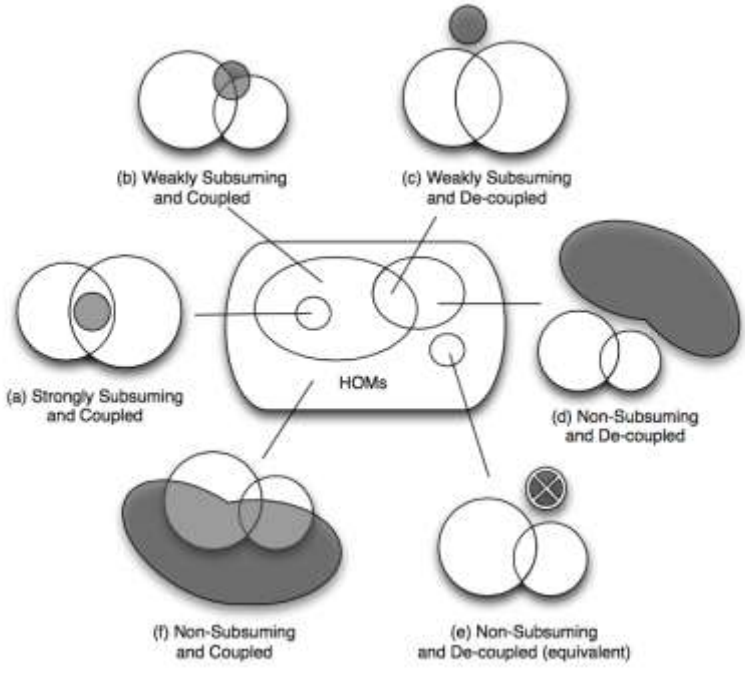
- Tìm kiếm tối ưu hóa đơn mục tiêu: Mục tiêu của phương pháp này nhằm tìm kiếm các HOM khó bị diệt hơn FOM mà nó được hình thành. Hướng tiếp cận này nhằm tìm kiếm các đột biến con (subsuming mutants) một cách rõ ràng.

- Tìm kiếm tối ưu hóa đa mục tiêu: Tìm kiếm các đột biến bậc cao dựa vào hai mục tiêu chính, khó bị diệt hơn các FOM tạo ra chúng và đồng thời ít thay đổi ngữ nghĩa của chương trình gốc nhất.

HOM có thể phân loại theo nghĩa "kết hợp" (coupled) và "tập con" (subsuming). Theo giải thuyết hiệu ứng liên kết, nếu một tập hợp các ca kiểm thử diệt các FOM cũng chứa các ca kiểm thử diệt HOM, ta gọi HOM đó là "coupled HOM", ngược lại "de-coupled HOM". Trong Hình 3, sơ đồ con là một "coupled HOM" nếu nó chứa vùng tô đen chồng lên các vùng không được tô đen, ví dụ như các sơ đồ con (a), (b), (f). Vì vùng tô đen của (c), (d) không chồng lên các vùng không tô, chúng là các "de-coupled HOM". Sơ đồ con (e) là một trường hợp đặc biệt của "de-coupled HOM" vì không có bất kỳ ca kiểm thử nào có thể diệt được đột biến bậc cao, nó không có sự trùng lặp, vì vậy, đây là một đột biến tương đương.

"Subsuming HOM" được định nghĩa là khó bị diệt hơn các FOM hình thành nên nó. Trong Hình 2, "subsuming HOM" được biểu diễn là vùng tô đen nhỏ hơn phần phần giao của 2 vùng không tô đen như trong các sơ đồ con (a), (b), (c); ngược lại, các đột biến bậc cao thể hiện trong (d), (e), (f) là non-subsuming. Hơn nữa. "subsuming HOM" cũng phân thành các đột biến con mạnh (strongly subsuming HOMs) và đột biến con yếu (weakly subsuming HOMs). Một ca kiểm thử diệt một đột biến con mạnh nó đảm bảo các thành phần đột biến bậc một của nó bị diệt hoàn toàn, ngược lại thì nó là đột biến yếu. Do đó, nếu vùng tô đen chỉ nằm trong phần giao của hai vùng không tô đen, nó là một Strong Subsuming HOM như trong hình (a). Ngược lại là các Weakly Subsuming HOM như trong hình (b) và (c).

Theo sự kết hợp của các subsuming và de-coupled HOM, sáu khả năng có thể xem xét đó là: (a) strong subsuming and coupled, (b) weakly subsuming and coupled, (c) weakly subsuming and de-coupled, (d) non subsuming and de-coupled, (e) non subsuming, de-coupled (là đột biến tương đương) và (f) non subsuming and coupled (trường hợp không sử dụng) như trong hình 2.



Hình 2. Phân loại các đột biến bậc cao [7]

Trong mục tiếp theo, chúng tôi sẽ trình bày chi tiết các nghiên cứu liên quan về kiểm thử đột biến bậc cao.

IV. CÁC NGHIÊN CỨU VỀ KIỂM THỬ ĐỘT BIẾN BẬC CAO

Kiểm thử bậc cao như đã đề cập ở trên là một giải pháp được sử dụng để nâng cao hiệu quả của kiểm thử đột biến. Thay vì sử dụng chỉ một sự thay đổi đơn giản như trong kiểm thử đột biến truyền thống, kiểm thử đột biến bậc cao sử dụng nhiều sự thay đổi phức tạp hơn để tạo ra các đột biến bằng cách áp dụng hai hoặc nhiều hơn các toán tử

đột biến. Có nhiều nghiên cứu về đột biến bậc cao, những nghiên cứu này chia làm hai nhóm: Kiểm thử đột biến bậc hai (Second Order Mutation Testing- SOM) và Kiểm thử đột biến bậc cao (High Order Mutation Testing- HOM).

A. Kiểm thử đột biến bậc hai

Năm 2008, kết quả nghiên cứu của Polo và cộng sự [18] cho rằng "Việc kết hợp các đột biến không làm giảm chất lượng của tập dữ liệu thử". Họ đề xuất ba thuật toán để tạo đột biến bậc hai. Với thuật toán LastToFirst, giả sử ta có danh sách đột biến gồm n đột biến bậc một. SOM được tạo bằng cách kết hợp FOM đầu tiên với FOM thứ n, FOM thứ 2 với FOM gần cuối (n-1) và tiếp tục như thế. Thuật toán thứ 2 là DifferentOperators, SOM được tạo bằng cách kết hợp hai FOM có toán tử đột biến khác nhau. Thuật toán cuối cùng, RandomMix, kết hợp ngẫu nhiên hai FOM để tạo ra SOM. Kết quả sử dụng phương pháp này cho thấy số đột biến SOM giảm 56% so với số đột biến FOM trong khi số đột biến tương đương giảm 74%.

Năm 2010, trên cơ sở các thuật toán được Polo và cộng sự đề xuất, Papadakis và Malevris [19] nghiên cứu và đề xuất năm chiến lược mới để kết hợp các FOM gồm: First2Last, SameNode, SameUnit, SU_F2Last và SU_DiffOp. Thuật toán SameNode kết hợp các đột biến bằng cách chọn các đột biến bậc 1 từ những khối cơ bản như nhau. SameUnit lựa chọn các cặp FOM từ các đơn vị chương trình tương tự trong khi SU_F2Last và SU_DiffOp lựa chọn các FOM trên cơ sở các thuật toán First2Last và DifferentOperators tương ứng. Họ thực hiện trên cả đột biến bậc 1 (strong mutants, Rand10%, Rand20%, Rand30%, Rand40%, Rand50%, Rand60%) và đột biến bậc 2 (RandomMix, DifferentOperator, First2Last, SameNode, SameUnit, SU_F2Last và SU_DiffOp) với cùng các chương trình kiểm thử. Theo kết quả nghiên cứu thực nghiệm, số đột biến do SOM sinh ra giảm 53.23% so với số đột biến được sinh bởi FOMs. So với đột biến mạnh (Strong mutation) Rand10% và SU_DiffOP trường hợp tốt nhất và xấu nhất theo nghĩa giảm số đột biến tương ứng là khoảng 89.99% và 27.60%. Trong khi đó, số đột biến tương đương giảm 73% so với FOMs. Đột biến mạnh so với Rand10% và Rand60% tỷ lệ giảm tương ứng trong trường hợp tốt nhất và xấu nhất là 89.91% và 39.18%.

Cũng trong năm 2010, Kintis và cộng sự [20] đề xuất hai nhóm cho kiểm thử đột biến bậc hai. Nhóm thứ nhất là nhóm chiến lược đột biến bậc hai (The Second Order Strategies) bao gồm chiến lược Ralaxed Dominator (RDomF) và Strict Doninator (SDomF). Nhóm thứ hai là chiến lược hỗn hợp (Hybrid Strategies). RDomF sử dụng một mức thư giản (relaxed level) của cặp đôi giống như cách sử dụng quan hệ đầu, cuối của quan hệ dây chuyền (Domination relation). RHomF bắt đầu bằng cách xây dựng một tập các nốt dây chuyền đầu và cuối (pre-n, post-n) cho mỗi nốt (n) của đồ thị điều khiển chương trình. Sau đó, RHomF kết nối đột biến đầu tiên của nốt n với đột biến đầu tiên không được chọn của nốt cuối (post -n), nốt thứ 2 với nốt đầu không được lựa chọn của nốt đầu (pre-n). Quá trình tiếp tục như vậy cho đến khi tất cả các đột biến của nốt n được dùng ít nhất một lần. SDomF sử dụng một mức cố định cho việc kết hợp bằng cách giới hạn việc lựa chọn đột biến giữa các cặp nốt được thống trị (dominated). Trong nhóm chiến lược hỗn hợp, các đột biến bậc một của khối thống trị cơ bản được kết hợp thành một đột biến bậc hai như trong chiến lược của SdomF và với một tập con được lựa chọn ngẫu nhiên của tập đột biến bậc một còn lại. 20% và 50% là tỷ lệ tương ứng của chiến lược Hdom (20%) và Hdom (50%) để lựa lựa chọn ngẫu nhiên các đột biến bậc một còn lại. Kết quả nghiên cứu của họ cho thấy số lượng đột biến tương đương giảm 78%. Tỷ lệ 86.8% và 65.5% là trường hợp tốt nhất và xấu nhất của SDomF và HdomF. Tuy nhiên, số đột biến sinh ra không được đề cập.

Trên cơ sở phân loại HOM được Jia và Harman giới thiệu, năm 2012 Omar và Ghost [21] sử dụng bốn hướng tiếp cận để tạo ra đột biến khó diệt trong lập trình hướng khía cạnh (Aspect -Oriented programing). Các hướng tiếp cận là: (1) chèn 2 lỗi vào một lớp cơ bản hoặc 2 lỗi vào một khía cạnh; (2) chèn 2 lỗi vào 2 lớp cơ bản khác nhau; (3) chèn 2 lỗi vào hai khía cạnh khác nhau; (4) chèn 1 lỗi vào 1 lớp và 1 lỗi vào 1 khía cạnh. Kết quả cho thấy rằng tỷ lệ của weakly subsuming -decoupled SOM và weakly subsuming -coupled SOM với tất cả đột biến sinh ra là không cao, thậm chí gần bằng 0, trong khi tỷ lệ của strong subsuming SOM là 7.3%. Kết quả cho thấy số đột biến SOMs sinh ra tăng cấp số mũ nhưng số đột biến tương đương giảm 40% so với FOM.

Gần đây, năm 2014, Madeyski và cộng sự [6] giới thiệu thuật toán JudyDiffOP và NeighPair để tạo đột biến bậc hai. JudyDiffOp là sự hiệu chỉnh thuật toán DifferentOperator của Polo với ý tưởng "tất cả các FOMs thành phần chỉ sử dụng duy nhất một lần để tạo ra một SOM". Thuật toán NeighPair được giới thiệu với ý tưởng "tạo SOM bằng việc kết hợp các FOM gần nhau nhất có thể". Sau đó, họ thực nghiệm 4 giải thuật RandomMix, Last2First (được đề xuất bởi Polo và cộng sự) cùng với JudyDiffOP và NeighPair trên các chương trình được kiểm thử là: Barbecue (7.413 dòng lệnh), Commons IO (16.283 dòng lệnh), Common Lang (48.507 dòng lệnh) và Common Math (80.023 dòng lệnh). Các giải thuật giảm số lượng đột biến SOM ít nhất là 50% so với FOM, giải thuật JudyDiffOP đạt kết quả tốt nhất 64%. Số đột biến tương đương giảm 57% và 66% khi sử dụng giải thuật JudyDiffOP, đây là giải thuật tốt nhất để giảm số đột biến tương đương. Giải thuật NeighPair không tốt theo nghĩa giảm số đột biến tương đương. Tuy nhiên, trong tất cả các thử nghiệm tỷ lệ đột biến SOM là cao hơn so với FOM [6].

Với tất cả các chiến lược đột biến bậc hai, số đột biến sinh ra giảm hơn 50% (ngoại trừ kết quả của Kintis và cộng sự) [24]. Trong khi số đột biến tương đương giảm 70% so với FOM. Bảng 5 thể hiện hiệu quả của các chiến lược SOM trong các nghiên cứu khác nhau đối với việc giảm số đột biến và số đột biến tương đương.

Bảng 5. So sánh kết quả của các nhóm tác giả sử dụng đột biến bậc hai

Tác giả	Số đột biến giảm (%)	Số đột biến tương đương giảm (%)
Polo et al [18]	56,06	74
Papadaki và Malevris [19]	53,23	73
Kintis et al [20]	Không đề cập	78
Omar và Ghost [21]	Không đề cập	40
Madeyki et al [23]	53,39	57

B. Kiểm thử đột biến bậc cao

Như đã trình bày ở trên, một HOM được gọi là subsuming HOM nếu nó khó bị diệt hơn các đột biến bậc 1 tạo nên nó. Jia và Harman đã đề xuất một số hướng tiếp cận để tìm ra các subsuming HOM [25-26] gồm: thuật toán tham lam (Greedy Algorithm), thuật toán di truyền (Genetic Algorithm) và thuật toán leo đồi (Hill-Climbing Algorithm). Để tìm ra subsuming HOMs hiệu quả hơn, họ cũng định nghĩa các khái niệm: Fragility value - giá trị ước lượng khả năng của một FOM hoặc một HOM bị diệt; Hàm thích nghi (Fitness function) - tỷ lệ dễ vỡ của một HOM so với một FOM [25-26]. Họ thực nghiệm với 10 chương trình C (14850 dòng lệnh và 35473 ca kiểm thử), kết quả cho thấy 67% HOM được sinh ra là khó bị diệt hơn so với FOM tạo nên chúng.

Với kiểm thử đột biến bậc cao dựa trên mô hình, Belli và cộng sự [27] giới thiệu kỹ thuật mới cho đột biến bậc một và bậc cao sử dụng 2 toán tử cơ bản là chèn và xoá (insertion and omission) trên mô hình dựa trên đồ thị. Theo kết quả thực nghiệm của họ số đột biến bậc cao (bậc 2 và 3) được tạo ra nhiều gấp bốn lần so với số đột biến bậc 1. Tỷ số đột biến tương đương của HOM so với số đột biến tạo ra là 55% so với số đột biến tương đương tạo ra bởi FOM.

Để so sánh số đột biến tương đương của FOM với HOM, Akinde [28] sử dụng phương pháp lấy mẫu để chọn ngẫu nhiên 250 FOM từ tập các FOM sinh ra và 200 SOM cùng 200 đột biến ngẫu nhiên từ 400 đột biến cho mỗi chương trình được kiểm thử. Tỷ lệ của FOM trên tổng số FOM là 19% trong khi tỷ lệ của HOMs là gần 0%. Theo hiểu biết tốt nhất của chúng tôi, đột biến bậc cao được tạo bởi hướng tiếp cận của Akinde là cực kỳ dễ diệt.

Omar cùng cộng sự năm 2013 và 2014 [29-30] giới thiệu một hướng mới cho việc phân loại HOM. Trong các phân loại của họ có 4 kiểu HOM. *Kết hợp toàn bộ* (Entirely Coupled HOM): nếu tập hợp các ca kiểm thử diệt HOM và phần giao của tất các tập hợp ca kiểm thử của FOM giống nhau; *Kết hợp một phần* (Partially coupled HOM): nếu có một sự khác nhau giữa tập các ca kiểm thử mà diệt HOM và phần giao của tất cả các tập ca kiểm thử diệt FOM; *Không kết đôi* (Decoupled HOM): nếu tập ca kiểm thử diệt HOM khác hoàn toàn với phần giao của tất cả các tập ca kiểm thử diệt HOM; *Subtle HOM*: nếu HOM có thể bị diệt bởi tập ca kiểm thử đưa ra. Mục đích chính của nghiên cứu là tìm ra Subtle HOM, là những HOM mà không thể bị diệt bởi tập hợp các ca kiểm thử diệt các FOM thành phần [29]. Các Subtle HOM có thể bị diệt bởi các ca kiểm thử có chất lượng cao hơn. Họ sử dụng các thuật toán di truyền, tìm kiếm cục bộ và tìm kiếm ngẫu nhiên để tìm ra các Subtle HOM và so sánh hiệu quả của các thuật toán đó. Kết quả của họ cho thấy khoảng 0.0012% các HOM sinh ra là Subtle HOM. Họ cũng kiểm tra và xác định rằng tất cả các Subtle HOM không phải là đột biến tương đương. Chúng khó bị diệt nhưng có thể bị diệt bởi những ca kiểm thử chất lượng hơn. Họ cũng chỉ ra rằng số lượng các HOM là cao hơn nhiều so với các FOM.

Để tìm ra đột biến bậc cao khó bị diệt và phản ánh lỗi thực tế hơn, Langdon và cộng sự [31-32] giới thiệu một hình thức mới của kiểm thử đột biến: kiểm thử đột biến bậc cao đa mục tiêu. Họ tin rằng có nhiều FOM dễ dàng bị diệt và không phản ánh lỗi thực tế. Họ đề xuất chèn vào các lỗi mà ngữ nghĩa gần với chương trình gốc thay vì chèn các lỗi cú pháp gần với chương trình gốc để tìm ra HOM mà có ngữ nghĩa tương tự với chương trình được kiểm thử. Họ đề xuất thuật toán NSGAII (Non -Dominated Sorting Genetic Algorithm) áp dụng lập trình di truyền và tối ưu hóa đa mục tiêu Pareto để tìm kiếm HOM với hai mục tiêu đưa ra là khó bị diệt và thay đổi ít ở mã nguồn. Kết quả chứng minh rằng với tiếp cận này có thể tìm ra HOM đại diện cho các lỗi phức tạp và khó bị diệt hơn FOM. Theo kết quả thực nghiệm của họ, số lượng đột biến tương đương giảm đáng kể nhưng số lượng đột biến thì tăng theo cấp số mũ của bậc đột biến. Với một chương trình C đơn giản, trong đó có 17 lượt sử dụng các toán tử so sánh (bao gồm 6 toán tử so sánh $<, <=, ==, !=, >=, >$), sẽ có 85 đột biến bậc 1, 3400 đột biến bậc 2, 85000 đột biến bậc 3, 1487500 đột biến bậc 4,...

Q. V. Nguyen và cộng sự trong những năm 2014-2016 đã đưa ra một cách phân loại các đột biến bậc cao nhằm bao phủ hết tất cả các trường hợp có thể có của các đột biến được sinh ra [9-14]. Trong cách phân loại của họ, có tất cả 11 nhóm đột biến. Trong đó nhóm đột biến "*Chất lượng cao và hợp lý*" (High quality and Reasonable HOM) [9-14] là nhóm các đột biến mà họ tập trung vào tìm kiếm. Các đột biến này được tạo ra bằng cách kết hợp các đột biến bậc một lại với nhau, khó bị diệt hơn các đột biến bậc một (*Hợp lý*) và tập các ca kiểm thử có thể diệt được các đột biến này sẽ đồng thời diệt được tất cả các đột biến bậc một kết hợp để tạo ra nó (*Chất lượng cao*). Đây là các đột biến mà có thể thay thế tất cả các đột biến bậc một của nó mà không làm giảm đi hiệu quả của kiểm thử đột biến. Điều này sẽ làm giảm số lượng đột biến được sinh ra, đồng thời giải quyết được vấn đề đột biến tương đương và mô tả được các lỗi thực tế của phần mềm. Tiếp theo, họ đã đề ra các hàm mục tiêu và các hàm thích nghi để từ đó áp dụng các thuật toán tối ưu hóa đa mục tiêu vào trong lĩnh vực kiểm thử đột biến bậc cao [9-14]. Họ đã sử dụng các thuật toán eMOEA, NSGAII, eNSGAII, NSGAIII, Random [9-14], và đề xuất thuật toán eNSGAII-DiffLOC để tìm kiếm các đột biến bậc cao chất lượng cao và hợp lý. eNSGAII-DiffLOC là thuật toán được họ đề xuất dựa trên sự thay đổi thuật toán eNSGAII với nguyên tắc "Không có quá một sự thay đổi (lỗi) trên một dòng lệnh". Sở dĩ họ lựa chọn thuật toán

eNSGA để thay đổi vì trong các đánh giá thực nghiệm thực tế, đây là thuật toán tốt nhất trong việc tạo ra các đột biến *Chất lượng cao và hợp lý* [14]. Nhóm nghiên cứu này đã xây dựng công cụ Judy[1] để sinh tự động các đột biến bậc cao từ bậc 2 đến bậc 15 bằng cách áp dụng các thuật toán tối ưu hóa đa mục tiêu dựa trên các hàm mục tiêu và thích nghi của họ. Những kết quả thực nghiệm đã chứng tỏ rằng phương pháp đề xuất đã có hiệu quả cao trong việc áp dụng kiểm thử đột biến nói chung, và kỹ thuật kiểm thử đột biến bậc cao nói riêng [9-14]. Cụ thể là có thể giảm chi phí tính toán thông qua việc giảm số lượng đột biến, tăng độ thực của lỗi (đột biến) và giúp hướng đến việc tạo ra các ca kiểm thử tốt hơn.

V. NHỮNG VẤN ĐỀ TỒN TẠI VÀ ĐỀ XUẤT HƯỚNG GIẢI QUYẾT

Từ các tổng hợp, phân tích và đánh giá trên, chúng ta có thể nhận thấy:

- Đối với đột biến bậc 2: Số lượng đột biến giảm đến 50% và số đột biến tương đương giảm đến 70% tuy nhiên số lượng đột biến vẫn còn cao và cần được tiếp tục cải tiến.

- Đối với đột biến bậc cao (bậc 2 trở lên): kết quả nghiên cứu của các nhóm tác giả cho thấy số đột biến tương đương giảm nhưng số đột biến sinh ra tăng theo cấp số mũ của bậc đột biến [31-32]. Ngược lại, với các nghiên cứu [9-14], số lượng đột biến sinh ra đã giảm đi rất nhiều so với số lượng đột biến bậc một, tuy nhiên số lượng đột biến khó bị diệt (có thể bao gồm cả đột biến tương đương) vẫn còn khá cao, chiếm khoảng 35% tổng số lượng đột biến được sinh ra.

Hơn nữa, vì sự hạn chế của thực nghiệm, các thách thức sau có thể ảnh hưởng đến kết quả.

Thứ nhất, việc lựa chọn toán tử đột biến: chẳng hạn, để giảm chi phí tính toán, thực nghiệm của Jia [16] sử dụng tập con gồm 77 toán tử cho chương trình C [33] để tạo đột biến bậc cao. Tập con toán tử được lựa chọn này cũng thuộc về 5 nhóm toán tử được đề xuất bởi Offutt [34-35], vì vậy nó là điển hình và được sử dụng rộng rãi bởi những nhà nghiên cứu khác. Hạn chế này có thể giải quyết trong tương lai bằng việc nghiên cứu mối quan hệ giữa đột biến bậc cao và toán tử đột biến.

Thứ hai, chất lượng của bộ dữ liệu kiểm thử, khi hàm thích nghi được tính toán với khả năng bị diệt của các đột biến (fragility values), chất lượng bộ dữ liệu thử thấp có thể ảnh hưởng đến kết quả. Việc kết hợp kiểm thử đột biến bậc cao với hướng kiểm thử đột biến đồng tiến hoá (co-evolutionary) của Adamopoulos và cộng sự [36] có thể giải quyết vấn đề này.

Thứ ba, sự tồn tại của đột biến tương đương, mặc dù vấn đề đề này được nhiều nhà nghiên cứu quan tâm, nhưng không hướng tiếp cận nào đảm bảo vừa hiệu quả vừa chính xác [37-39]. Hàm thích nghi cho việc tìm kiếm đột biến bậc cao được thiết kế để lọc ra các đột biến tương đương tiềm tàng. Với bộ dữ liệu thử kém chất lượng, một số đột biến khó diệt (stubborn decoupled) có thể được xử lý không chính xác như đột biến tương đương.

Xuất phát từ những vấn đề tồn tại nên trên, một số định hướng nghiên cứu cho kiểm thử bậc cao được chúng tôi đề xuất dưới đây.

A. Xây dựng mô hình lỗi trong đột biến bậc cao dựa vào học máy

Nghiên cứu xây dựng một mô hình lỗi nhằm đưa ra các lỗi đặc trưng mà lập trình viên thường phạm phải chứ không phải đưa ra tất cả các lỗi tiềm năng có thể có. Tuỳ vào thói quen lập trình, lĩnh vực áp dụng để đưa ra cách phân loại, cách kết hợp các loại lỗi khác nhau. Áp dụng học máy để tìm mô hình lỗi cho từng loại ngôn ngữ hay lĩnh vực ứng dụng. Hướng nghiên cứu này phù hợp với kiểm thử đột biến bậc cao bởi vì đột biến bậc cao là sự kết hợp của từ 2 sự thay đổi trong một đột biến.

B. Sinh dữ liệu thử dựa trên mô hình lỗi

Một phần mềm muốn có chất lượng tốt, phần mềm đó phải được kiểm thử trước khi sử dụng, nghĩa là phải được thực thi trên dữ liệu thử. Việc sinh dữ liệu thử từ mọi miền vào của chương trình là không khả thi và tốn kém, vì vậy chúng ta phải tạo ra tập dữ liệu thử hiệu quả theo nghĩa số lượng dữ liệu thử ít nhất mà phát hiện được nhiều lỗi nhất. Một bộ dữ liệu thử như thế phải được cập nhật theo hướng ngày càng phát hiện được nhiều lỗi, nghĩa là phải có khả năng "học". Với một mô hình lỗi, đầu vào là tập dữ liệu thử, đầu ra là kết quả các lỗi được phát hiện bởi tập dữ liệu thử đó. Nếu dữ liệu thử chưa chất lượng, sẽ tiến hành cải tiến cho đến khi bộ dữ liệu thử đạt yêu cầu. Sử dụng mô hình lỗi để huấn luyện bộ dữ liệu thử cũng là một hướng nghiên cứu đáng quan tâm hiện nay đặc biệt là dữ liệu thử diệt được các đột biến bậc cao nhằm chỉ ra các lỗi thực và lỗi phức tạp.

C. Đồng tiến hoá đột biến bậc cao và ca kiểm thử

Đồng tiến hoá là một hướng tiếp cận để tối ưu hoá đồng tiến trong đó hai hoặc nhiều cư dân ứng viên đồng tiến cùng nhau. Có hai trường hợp xảy ra. Thứ nhất, đó có thể là một quá trình hợp tác, mô phỏng hành vi cộng sinh trong tiến hoá tự nhiên. Thứ hai, đó là quá trình cạnh tranh, mô phỏng hành vi săn mồi của sự thích ứng và tiến bộ đồng tiến hoá.

Với kiểm thử đột biến bậc cao, hai ứng viên có thể là đột biến và ca kiểm thử. Sử dụng lý thuyết đồng tiến hoá là xây dựng đột biến sao cho trốn tránh được nhiều ca kiểm thử nhất đồng thời xây dựng các ca kiểm thử sao cho diệt được nhiều đột biến nhất. Chúng phù hợp với mô hình đồng tiến hoá cạnh tranh, có thể dùng để phát triển tập đột biến bậc cao khó bị diệt và đồng thời thiết lập các ca kiểm thử chất lượng tốt được điều chỉnh làm lộ ra các lỗi tinh vi và khó bị diệt.

D. Khám phá các kiểu đột biến bậc cao

Trong nghiên cứu của nhóm tác giả Jia [17] đã đưa ra cách phân loại đột biến bậc cao gồm 6 nhóm và nhóm Q. V. Nguyen và cộng sự [9-14] đưa ra 11 nhóm. Tuy vậy, các tác giả chỉ tập trung vào một số nhóm nhỏ nhất định, chẳng hạn nhóm tác giả Jia tập trung vào nhóm đột biến mạnh (Strong High Order Mutation - SHOM) trong khi nhóm Q. V. Nguyen tập trung vào nhóm đột biến "*Chất lượng cao và hợp lý*" (High quality and Reasonable HOM). Vì vậy, có thể tiếp tục nghiên cứu cụ thể hơn về các nhóm còn lại hoặc đề xuất các phân loại mới cho đột biến bậc cao nhằm làm giảm số lượng đột biến được sinh ra, hoặc giảm đột biến tương đương và mô tả được các lỗi thực tế của phần mềm.

VI. KẾT LUẬN

Kiểm thử đột biến cao có thể thực tế hoá khi được thực thi như một quy trình tìm kiếm để tìm ra các đột biến phù hợp từ tập không gian các đột biến. Giảm số lượng đột biến mà vẫn không làm ảnh hưởng đến hiệu quả của hoạt động kiểm thử đồng thời cải thiện chất lượng tập dữ liệu thử nhằm phát hiện được các lỗi thực và khó bị diệt là mục tiêu hướng đến của kiểm thử đột biến bậc cao. Các nghiên cứu và kết quả thực nghiệm đã chứng tỏ đột biến bậc cao phản ánh được các lỗi thực và khó bị diệt hơn đột biến truyền thống. Trong bài báo này, chúng tôi đã tổng hợp và phân tích những nghiên cứu và hiệu quả ứng dụng của đột biến bậc cao nhằm cung cấp một cách nhìn đầy đủ cho lĩnh vực nghiên cứu này.

Từ phân tích các hạn chế hiện tại của kiểm thử đột biến bậc cao, chúng tôi đã vạch ra các hướng nghiên cứu nhằm nâng cao hiệu quả kiểm thử của đột biến bậc cao cũng như khả năng ứng dụng vào thực tế.

TÀI LIỆU THAM KHẢO

[1] Lu Luo, *Technology Maturation and Research Strategy*, Institute for Software Rearch International, Carnegie Mellon University Pittsburgh, PA 15232 USA.

[2] R. DeMillo, R. Lipton, and F. Sayward, "*Hints on Test Data Selection: Help for Practicing for Programmer,"* IEEE computer, no. 11, pp. 34-41, 1978.

[3] T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, *"Mutation Analysis*," Atlanta, Georgia, 1979.

[4] R. Geist, A. J. Offutt, and F. C. Harris, "Estimation and Enhancement of Real-Time Software Reliability Through Mutation Analysis," IEEE Transactions on Computers, vol. 41, no. 5, p. 550-558, May 1992.

[5] S. Eldh, S. Punnekkat, H. Hansson, and P. J¨onsson. Component Testing Is Not Enough - A Study of Software Faults in Telecom Middleware. In Proceedings of the 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software (TestCom'07) and the 7th International Workshop (FATES'07), Tallinn, Estonia, 26-29 June 2007.

[6] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. IEEE Transactionson Software Engineering, 40 (1):23-42, 2014

[7] A. J. Offutt, "The Coupling Effect: Fact or Fiction," ACM SIGSOFT Software Engineering Notes, Vols. vol. 14, no. 8, p. 131-140, December 1989.

[8] Y Jia, M Harman - An analysis and survey of the development of mutation testing, IEEE transactions on software engineering, 37 (2011) 649-678.

[9] Quang Vu Nguyen and L. Madeyski. *Problems of mutation testing and higher order mutation testing*. Advanced Computational Methods for Knowledge Engineering, Advances in Intelligent Systems and Computing, 282, 2014.

[10] Quang Vu Nguyen and L. Madeyski. *Searching for strongly subsuming higher order mutants by applying multiobjective optimization algorithm*. Advanced Computational Methods for Knowledge Engineering , Advances in IntelligentSystemsandComputing, 358:391-402, 2015.

[11] Quang Vu Nguyen and L. Madeyski. *Higher order mutation testing to drive development of new test cases: an empirical comparison of three strategies*. Lecture Notes in Computer Science, vol. 9621, 2016.

[12] Quang Vu Nguyen and L. Madeyski. *On the relationship between the order of mutation testing and the properties of generated higher order mutants*. Lecture Notes in Computer Science, 2016.

[13] Quang Vu Nguyen and L. Madeyski. *Empirical evaluation of multi-objective optimization algorithms searching for higher order mutants*. Cybernetic and Systems: An International Journal, 47 (1-2), 2016.

[14] Quang Vu Nguyen and L. Madeyski. *Addressing mutation testing problems by applying multi-objective optimization algorithms and higher order mutation*. Journal of Intelligent & Fuzzy Systems, vol. 32, No 2, pp. 1173-1182, 2017.

[15] R. Purushothaman and D. E. Perry. Toward Understanding the Rhetoric ofSmall Source Code Changes.

IEEE Transactions on Software Engineering,31(6):511-526, 2005.

[16] Yue Jia, "Higher Order Mutation Testing", Department of Compuer Science, University College London University of London, 2013

[17] M. Harman, Y. Jia, and W. B. Langdon. A manifesto for higher order mutation testing. Third International Conf. on Software Testing, Verification, and Validation Workshops, 2010.

[18] M. Polo, M. Piattini, and I. Garcia-Rodriguez. Decreasing the cost of mutation testing with second-order mutants. Software Testing, Verification, and Reliability, 19 (2):111-131, 2008.

[19] M. Papadakis and N. Malevris. An empirical evaluation of the first and second order mutation testing strategies. Proceedings of the 2010 Third International Conferenceon Software Testing, Verification, and Validation Workshops, ser. ICSTW'10, IEEE Computer Society, pages 90-99, 2010.

[20] M. Kintis, M. Papadakis, and N. Malevris. Evaluating mutation testing alternatives: A collateral experiment. Proceedings 17th Asia Pacific Software Engineering. Conference (APSEC), 2010.

[21] E. Omar and S. Ghosh. An exploratory study of higher order mutation testing in aspect-oriented programming. IEEE 23rd International Symposium on Software Reliability Engineering, 2012.

[22] E. Omar and S. Ghosh. An exploratory study of higher order mutation testing in aspect-oriented programming. IEEE 23rd International Symposium on Software Reliability Engineering, 2012.

[23] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala. *Overcoming the equivalent mutant problem*: *A systematic literature review and a comparative experiment of second order mutation*. IEEE Transactionson Software Engineering, 40 (1):23-42, 2014.

[24] M. Kintis, M. Papadakis, and N. Malevris. Evaluating mutation testing alternatives: A collateral experiment. Proceedings 17th Asia Pacific Software Engineering. Conference (APSEC), 2010.

[25] Y. Jia and M. Harman. Higher order mutation testing. Information and Software Technology, 51:1379-1393, 2009.

[26] Y. Jia and M. Harman. Constructing subtle faults using higherordermutationtesting. Proc. Eighth Int'l Working Conf. Source Code Analysis and Manipulation, 2008.

[27] F. Fevzi Belli, N. Guler, A. Hollmann, and E. Suna, G.and Esra Yildiz. Model-based higher-order mutation analysis. Series Communications in Computer and Information Science, Advances in Software Engineering, 117:164-173, 2010.

[28] A. O. Akinde. Using higher order mutation for reducing equivalent mutants in mutation testing. Asian Journal Of Computer Science And Information Technology, 2 (3):13-18, 2012.

[29] E. Omar, S. Ghosh, and D. Whitley. Constructingsubtle higher order mutants for java and aspectj programs. International Symposium on Software Reliability Engineering, pages 340-349, 2013.

[30] E. Omar, S. Ghosh, and D. Whitley. Comparing search techniques for finding subtle higher order mutants. Proceedings of the 2014 Annual conference on Genetic and Evolutionary computation, pages 1271-1278, 2014.

[31] W. B. Langdon, M. Harman, and Y. Jia. Multi-objective higher order mutation testing with genetic programming. Proceedings Fourth Testing: Academic and Industrial Conference Practice and Research, 2009.

[32] W. B. Langdon, M. Harman, and Y. Jia. Efficient multiobjective higher order mutation testing with genetic programming. The Journal of Systems and Software, 83, 2010.

[33] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford. Design of Mutant Operators for the C Programming Language. Technique Report SERC-TR-41-P, Purdue University, West Lafayette, Indiana, March 1989.

[34] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An Experimental Determination of Sufficient Mutant Operators. ACM Transactions on Software Engineering and Methodology, 5(2):99-118, April 1996.

[35] A. J. Offutt, G. Rothermel, and C. Zapf. An Experimental Evaluation of Selective Mutation. In Proceedings of the 15th International Conference on Software Engineering (ICSE'93), pages 100-107, Baltimore, Maryland, May 1993. IEEE Computer Society Press.

[36] K. Adamopoulos, M. Harman, and R. M. Hierons. How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'04), volume 3103 of LNCS, pages 1338-1349, Seattle, Washington, USA, 26th-30th, June 2004. Springer.

[37] R. M. Hierons, M. Harman, and S. Danicic. Using Program Slicing to Assist in the Detection of Equivalent Mutants. Software Testing, Verification and Reliability, 9(4):233-262, December 1999.

[38] A. J. Offutt and W. M. Craft. Using Compiler Optimization Techniques to Detect Equivalent Mutants. Software Testing, Verification and Reliability, 4(3):131-154, September 1994.

[39] A. J. Offutt and J. Pan. Automatically Detecting Equivalent Mutants and Infeasible Paths. Software Testing, Verification and Reliability, 7(3):165-192, September 1997.

HIGHER ORDER MUTATION TESTING: THE EFFECT AND LIMITED PROLEMS

ABSTRACT*: Mutation testing (also called first order mutation testing) has been introduced as a technique to assess the quality of test cases. It is a highly automated and useful technique for evaluating the fault-finding effectiveness of tests, and many researchers are now involved in mutation testing research. However, there are three main challenges: (1) a large number of mutants (leading to a very high computational cost); (2) generated mutants can really reflect faults; and (3) equivalent mutant problems. There are many different approaches which have been proposed for overcoming the above-mentioned problems of mutation testing. Each solution has its own advantages and disadvantages. Among them, higher order mutation testing, which was first introduced in 2009, is the only approach abling to address of the three mutation testing problems, and the only one to deal with the problem of realism of injected defects. In this paper, we consider the effectiveness as well as the development of the methods in area of higher order mutation testing. Based on that, we evaluate the existing problems and discuss the future research directions.*

Keywords*: Software testing; Mutation Testing; Higher Order Mutation Testing; Limitations of Mutation Testing*.