ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

NGUYỄN THỊ YÊN

CÁC KỸ THUẬT TRONG KIỂM THỬ DÒNG DỮ LIỆU TĨNH

Chuyên ngành: Kỹ thuật phần mềm
Mã số: 6048103

TÓM TẮT LUẬN VĂN THẠC SĨ

Hà Nội - 2016

LỜI CAM ĐOAN

Tôi xin cam đoan:

Những kết quả nghiên cứu được trình bày trong luận văn là hoàn toàn trung thực, của tôi, không vi phạm bất cứ điều gì trong luật sở hữu trí tuệ và pháp luật Việt Nam. Nếu sai, tôi xin hoàn toàn chịu trách nhiệm trước pháp luật.

TÁC GIẢ LUẬN VĂN

Nguyễn Thị Yên

MỤC LỤC

DANH MỤC CÁC KÝ HIỆU, CHỮ VIẾT TẮT

TT	Viết tắt	Đầy đủ	Diễn giải
1	TBFV	Testing - Based Formal Verification	Kỹ thuật chứng minh hình thức dựa trên kiểm thử
2	RUP	Rational Unified Process	Qui trình phát triển phần mềm
3	DU-path	Definition Use path	Đường dẫn định nghĩa sử dụng
4	FSF	Functional Scenario Form	Hình thức kịch bản chức năng
5	SOFL	Structured Object-Oriented Formal Language	Ngôn ngữ hình thức hướng đối tượng cấu trúc

DANH MỤC CÁC HÌNH

Trang

MỞ ĐẦU

Kỹ thuật phân tích dòng dữ liệu tĩnh giúp phát hiện sớm các lỗi tiềm ẩn trong chương trình từ đó giảm thiểu kinh phí, công sức cho kiểm thử phần mềm. Từ những lý do trên nên em đã chọn đề tài: ***"Các kỹ thuật trong kiểm thử dòng dữ liệu tĩnh".***

Mục tiêu của đề tài: Nghiên cứu Tổng quan về kiểm thử phần mềm để nắm những kiến thức cơ bản phục vụ cho các nghiên cứu tiếp theo. Sau đó nghiên cứu Tổng quan về các phương pháp kiểm thử phần mềm và kiểm thử dòng dữ liệu tĩnh. Tiếp theo nghiên cứu ứng dụng Logic Hoare trong kiểm thử phần mềm, cụ thể: nghiên cứu kết hợp Logic Hoare với kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu và áp dụng kỹ thuật kết hợp này vào kiểm thử một đoạn chương trình.

Cấu trúc của luận văn được chia thành 3 chương cụ thể như sau:

Chương 1: Tổng quan về Kiểm thử phần mềm và kiểm thử phần mềm tĩnh

Chương 2: Phương pháp kiểm thử dòng dữ liệu tĩnh trong kiểm thử phần mềm

Chương 3: Ứng dụng Logic Hoare trong kiểm thử phần mềm

Để hoàn thành được luận văn, em xin được gửi lời cảm ơn tới các thầy cô trong Khoa Công nghệ thông tin - Trường Đại học Công nghệ đã tận tình giảng dạy, cung cấp nguồn kiến thức quý giá trong suốt quá trình học tập.

Đặc biệt em xin chân thành cảm ơn thầy giáo TS.Đặng Văn Hưng đã tận tình hướng dẫn, góp ý, tạo điều kiện cho em hoàn thành luận văn này.

Chương 1
TỔNG QUAN VỀ KIỂM THỬ PHẦN MỀM VÀ KIỂM THỬ TĨNH

Trong chương này trình bày những kiến thức cơ bản liên quan đến Kiểm thử phần mềm như định nghĩa kiểm thử phần mềm, các mức kiểm thử phần mềm… Đồng thời cũng trình bày khái quát về Kiểm thử tĩnh.

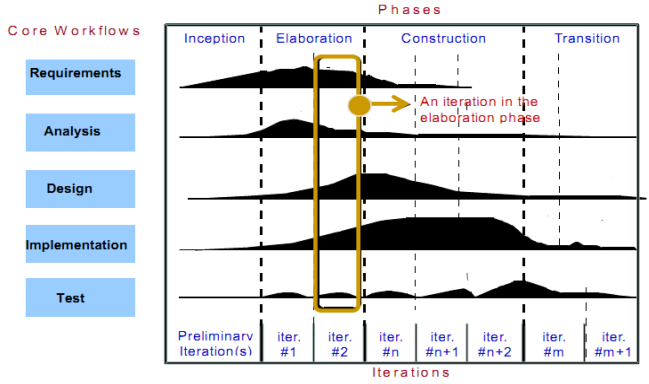
1.1. Khái quát về Kiểm thử phần mềm

1.1.1. Định nghĩa về Kiểm thử phần mềm

Kiểm thử phần mềm là qui trình nhằm đảm bảo chất lượng phần mềm. Kiểm thử phần mềm hướng tới việc chứng minh phần mềm không có lỗi.

Mục đích của kiểm thử phần mềm là phát hiện lỗi càng sớm càng tốt, và đảm bảo rằng những lỗi này phải được sửa. Lỗi được hiểu là phần mềm không hoạt động đúng như đặc tả của nó.

1.1.2. Qui trình phát triển phần mềm RUP [1]



Hình 1.1: Qui trình phát triển phần mềm RUP

1.1.3. Các mức kiểm thử phần mềm [1]

- Kiểm thử đơn vị (Unit Testing)
- Kiểm thử module (Module Testing)
- Kiểm thử tích hợp (Intergration Testing)
- Kiểm thử hệ thống (System Testing)
- Kiểm thử chấp nhận (Acceptance Testing)

1.1.4. Ca kiểm thử và các phương pháp thiết kế ca kiểm thử

Mỗi ca kiểm thử chứa các thông tin cần thiết để kiểm thử thành phần phần mềm theo một mục tiêu xác định. Thường ca kiểm thử gồm bộ ba thông tin {dữ liệu đầu vào, trạng thái của thành phần phần mềm, dữ liệu đầu ra kỳ vọng} [1].

Các phương pháp thiết kế ca kiểm thử

Bất kỳ sản phẩm kỹ thuật nào (phần mềm không phải là ngoại lệ) đều có thể được kiểm thử bởi một trong hai chiến lược [1]:

- Chiến lược kiểm thử hộp đen (Black box testing):
- Chiến lược Kiểm thử hộp trắng (White box testing):

1.2. Khái quát về Kiểm thử tĩnh

1.2.1. Định nghĩa về Kiểm thử tĩnh

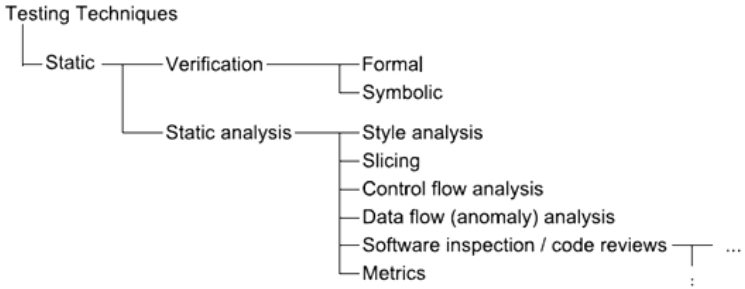
Kiểm thử tĩnh là một hình thức của kiểm thử phần mềm mà không chạy chương trình (hoặc phần mềm) được kiểm thử. Điều này ngược với kiểm nghiệm động. Thường thì nó không kiểm thử chi tiết mà chủ yếu kiểm tra tính đúng đắn của code (mã lệnh), thuật toán hay tài liệu [1].

1.2.2. Phân loại các kỹ thuật kiểm thử tĩnh

Các kỹ thuật kiểm thử tĩnh không tạo ra các ca kiểm thử vì không chạy chương trình được kiểm thử [17]. Các kỹ thuật kiểm thử tĩnh có thể được chia thành hai nhóm kỹ thuật:

- Nhóm kỹ thuật kiểm thử kiểm tra (verification tests);
- Nhóm kỹ thuật kiểm thử phân tích (analysis tests).

Phân loại các kỹ thuật kiểm thử tĩnh được tổng hợp trong Hình 1.4.



Hình 1.4: Phân loại các kỹ thuật kiểm thử tĩnh

1.2.3. Sơ lược về các kỹ thuật kiểm thử tĩnh

➢ ***Phân tích style*** là một kỹ thuật mà kiểm tra mã nguồn xem có thực thi theo đúng mong muốn không. Phân tích style là một phần của các

hệ thống nhúng; hơn nữa phân tích tĩnh là một lựa chọn tốt để duy trì vì vậy cũng được gọi là *các hệ thống legacy*.

➢ ***Phân tích dòng điều khiển (Control flow analysis)***: kiểm tra code để phát hiện các bất thường (anomalies). Mục tiêu của kỹ thuật là phát hiện code mà không được thực thi (*dead code*) và phát hiện các vòng lặp mà không thể thoát khỏi vòng lặp [18].

➢ ***Phân tích luồng (bất thường) dữ liệu* hoặc *phát hiện bất thường luồng dữ liệu*** được sử dụng để phát hiện các thành phần của chương trình mà đi lệch với mong muốn.

➢ ***Phân tích luồng dữ liệu và luồng điều khiển*** có thể được kết hợp với các công cụ mà duyệt tự động code cho việc phát hiện các bất thường. Chúng có thể được dựa trên vài kỹ thuật khác và có thể cung cấp hỗ trợ giao diện trực quan [22].

1.3. Kết luận

Kiểm thử tĩnh (phân tích tĩnh) là một hình thức của kiểm thử phần mềm. Tuy nhiên kỹ thuật kiểm thử này không chạy chương trình được kiểm thử.

Chương 2
PHƯƠNG PHÁP KIỂM THỬ DÒNG DỮ LIỆU TĨNH TRONG KIỂM THỬ PHẦN MỀM

Trong chương này tác giả sẽ tập trung vào nghiên cứu và trình bày các phương pháp kiểm thử dòng dữ liệu tĩnh.

2.1. Phương pháp kiểm thử dòng dữ liệu tĩnh

2.1.1. Ý tưởng của phương pháp

Mỗi chương trình/đơn vị chương trình là chuỗi các hoạt động gồm nhận các giá trị đầu vào, thực hiện các tính toán, gán giá trị mới cho các biến (các biến cục bộ và toàn cục) và cuối cùng là trả lại kết quả đầu ra như mong muốn. Khi một biến được khai báo và gán giá trị, nó phải được sử dụng ở đâu đó trong chương trình. Việc sử dụng biến này có thể trong các câu lệnh tính toán hoặc trong các biểu thức điều kiện. Nếu biến này không được sử dụng ở các câu lệnh tiếp theo thì việc khai báo biến này là không cần thiết. Hơn nữa, cho dù biến này có được sử dụng thì tính đúng đắn của chương trình chưa chắc đã đảm bảo vì lỗi có thể xảy ra trong quá trình tính toán hoặc trong các biểu thức điều kiện. Để giải quyết vấn đề này, phương pháp kiểm thử dòng dữ liệu xem đơn vị chương trình gồm các đường đi tương ứng với các dòng dữ liệu nơi mà các biến được khai báo, được gán

giá trị, được sử dụng để tính toán và trả lại kết quả mong muốn của đơn vị chương trình ứng với đường đi này.

Với mỗi đường đi, chúng ta sẽ sinh một ca kiểm thử để kiểm tra tính đúng đắn của nó. Quá trình kiểm thử dòng dữ liệu được chia thành hai pha riêng biệt: kiểm thử dòng dữ liệu tĩnh (static data flow testing) và kiểm thử dòng dữ liệu động (dynamic data flow testing). Với kiểm thử dòng dữ liệu tĩnh, chúng ta áp dụng các phương pháp phân tích mã nguồn mà không cần chạy chương trình/đơn vị chương trình nhằm phát hiện các vấn đề về khai báo, khởi tạo giá trị cho các biến và sử dụng chúng. Chi tiết về vấn đề này sẽ được trình bày trong phần tiếp theo. Với kiểm thử dòng dữ liệu động, chúng ta sẽ chạy các ca kiểm thử nhằm phát hiện các lỗi tiềm ẩn mà kiểm thử tĩnh không phát hiện được.

2.1.2. Các vấn đề bất thường trong dòng dữ liệu

Các vấn đề bất thường về dòng dữ liệu có thể được phát hiện bằng phương pháp kiểm thử dòng dữ liệu tĩnh. Theo Fosdick và Osterweil [4], các vấn đề này được chia thành ba loại như sau:

- Gán giá trị rồi gán tiếp giá trị (Loại 1)
- Chưa gán giá trị nhưng được sử dụng (Loại 2)
- Đã được khai báo và gán giá trị nhưng không được sử dụng (Loại 3)

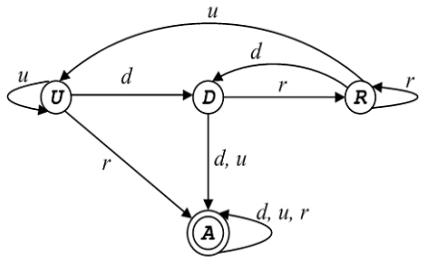
Huang [5] đã giới thiệu một phương pháp để xác định những bất thường trong việc sử dụng các biến dữ liệu bằng cách sử dụng sơ đồ chuyển trạng thái ứng với mỗi biến dữ liệu của chương trình. Các thành phần của sơ đồ chuyển trạng thái của một chương trình ứng với mỗi biến gồm:

Các trạng thái, gồm:

- U: biến chưa được gán giá trị
- D: biến đã được gán giá trị nhưng chưa được sử dụng
- R: biến đã được sử dụng
- A: trạng thái lỗi

• Các hành động, gồm:

- d: biến được gán giá trị
- r: biến được sử dụng
- u: biến chưa được gán giá trị hoặc được khai báo lại và chưa được gán giá trị.



Hình 2.3: Sơ đồ chuyển trạng thái của một biến

Các vấn đề với dòng dữ liệu thuộc loại 1 ứng với trường hợp dd xảy ra trong sơ đồ chuyển trạng thái. Các vấn đề thuộc loại 2 ứng với trường hợp ur và loại 3 ứng với trường hợp du. Để phát hiện các vấn đề này, chúng ta sẽ tiến hành xây dựng sơ đồ chuyển trạng thái ứng với mỗi biến như Hình 2.3. Nếu trạng thái A xuất hiện thì chương trình có vấn đề về dòng dữ liệu.

2.1.3. Phương pháp kiểm thử dòng dữ liệu tĩnh

Khái niệm của kiểm thử dòng dữ liệu tĩnh cho phép người kiểm thử xác định các biến chạy qua chương trình, giúp người kiểm thử đảm bảo chắc chắn rằng không xảy ra một lỗi mà được miêu tả ở trên.

Kiểm thử dòng dữ liệu tĩnh có thể được xem xét như là một hình thức của kiểm thử cấu trúc (structual testing): ngược với kiểm thử chức năng (kiểm thử hộp đen). Các kỹ thuật kiểm thử cấu trúc yêu cầu người kiểm thử phải truy cập các chi tiết cấu trúc của chương trình. Các biến được định nghĩa và được sử dụng tại các điểm khác nhau bên trong một chương trình; kiểm thử dòng dữ liệu tĩnh cho phép người kiểm thử vẽ đồ thị thay đổi các giá trị của các biến bên trong một chương trình.

Có hai hình thức chính của kiểm thử dòng dữ liệu tĩnh: đầu tiên, được gọi là kiểm thử định nghĩa/sử dụng (define/use), sử dụng số lượng các luật đơn giản và các độ đo kiểm thử (test coverage metrics); thứ hai sử dụng "program slices" – các đoạn của một chương trình.

2.2. Kết luận

Chúng ta thấy rằng kỹ thuật kiểm thử dòng dữ liệu tĩnh được sử dụng ở mức kiểm thử đơn vị và được sử dụng để phát hiện những lỗi trong việc khai báo, sử dụng các biến trong một chương trình/đơn vị chương trình. Nó giúp cho chương trình/đơn vị chương trình chạy ổn định và đưa ra kết quả mong muốn.

Trong [12], tác giả đã chỉ ra một phương pháp phát hiện lỗi trong một chương trình/đơn vị chương trình, đó là sử dụng Logic Hoare trong kiểm thử phần mềm. Chương tới, chúng tôi sẽ trình bày chi tiết phương pháp phát hiện lỗi này.

Chương 3
ỨNG DỤNG LOGIC HOARE TRONG KIỂM THỬ PHẦN MỀM

Chương này tác giả tập trung vào trình bày phương pháp ứng dụng Logic Hoare trong kiểm thử phần mềm, cụ thể tác giả trình bày phương pháp kiểm thử kết hợp giữa Logic Hoare với kỹ thuật kiểm thử dựa vào kịch bản dòng dữ liệu.

3.1. Đặt vấn đề

Với phạm vi của đề tài nghiên cứu, trong chương này tác giả sẽ chỉ tập trung trình bày cách ứng dụng Logic Hoare trong kiểm thử phần mềm. Cụ thể là tác giả trình bày phương pháp kiểm thử kết hợp giữa Logic Hoare với kỹ thuật kiểm thử dựa vào kịch bản dòng dữ liệu.

3.2. Tổng quan về Logic Hoare

Logic Hoare sử dụng các bộ ba Hoare (Hoare Triples) để suy diễn về sự chính xác của chương trình. Một bộ ba Hoare có dạng *{P}S{Q}*, ở đây *P* là precondition, *Q* là postcondition, và *S* là các lệnh của hàm. Ý nghĩa của bộ ba *{P}S{Q}* (ở dạng chính xác toàn bộ) là nếu chúng ta bắt đầu ở trạng thái *P* là true và thực thi *S*, khi đó *S* sẽ kết thúc ở trạng thái *Q* là true. Và chúng ta có thể suy diễn theo hướng ngược lại.

Chúng ta có thể định nghĩa một hàm có precondition yếu nhất cùng với postcondition nào đó cho các phép gán, chuỗi các lệnh, và nếu các lệnh như sau:

wp(x := E, P) = *[E/x]P*
wp(S; T, Q) = *wp(S; wp(T, Q))*
*wp(*if *B* then *S* else *T, Q)* = *B => wp(S, Q) && ¬B => wp(T, Q)*

Để chứng minh chính xác cục bộ của các lặp ở dạng *While b do S*, chúng ta xây dựng một *I* không biến đổi thỏa mãn các điều kiện dưới đây:

P => I: Khởi tạo không biến đổi là true.

{Inv && B} S {Inv}: Mỗi khi thực thi của lặp duy trì không biến đổi.

(Inv && ¬B)=>Q: Không biến đổi và lặp thoát điều kiện postcondition.

Chúng ta có thể chứng minh chính xác toàn bộ bằng cách xây dựng một hàm biến đổi giá trị nguyên (integer) *v* mà đáp ứng các điều kiện dưới đây:

Inv && B => v > 0: Nếu chúng ta vào body của lặp (tức là điều kiện lặp *B* đánh giá là true) và không biến đổi giữ nguyên khi đó *v* phải dương.

{Inv && B && v = V} S {v < V}: Giá trị của hàm biến đổi *v* giảm mỗi lần body của lặp thực thi (ở đây *V* là hằng số).

3.3. Ứng dụng Logic Hoare trong kiểm thử phần mềm

Chúng ta biết rằng Logic Hoare được sử dụng để chứng minh sự chính xác của các chương trình trong khi đó kiểm thử là một cách sử dụng trong thực tế để phát hiện các lỗi trong các chương trình. Tuy nhiên việc sử dụng Logic Hoare hiếm khi được áp dụng trong thực tế và kiểm thử cũng khó phát hiện tất cả các lỗi xuất hiện trong các chương trình. Do vậy, trong phần này tác giả sẽ trình bày một kỹ thuật kết hợp Logic Hoare với kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu để nâng cao khả năng phát hiện lỗi cho kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu.

3.3.1. Sơ lược kỹ thuật kiểm thử dựa vào kịch bản dòng dữ liệu

Kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu là phương pháp kiểm thử dựa trên đặc tả sử dụng precondition và postcondition trong việc tạo ra ca kiểm thử [16]. Áp dụng nguyên lý "chia và trị" thì kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu coi đặc tả là *các kịch bản dòng dữ liệu* được tách biệt và tạo ra các tập kiểm thử và phân tích các kết quả kiểm thử dựa trên các kịch bản dòng dữ liệu.

Một kịch bản dòng dữ liệu ở dạng đặc tả kiểu pre-post là một biểu thức logic mà thể hiện một cách rõ ràng rằng điều kiện gì được sử dụng để ràng buộc đầu ra khi đầu vào thỏa mãn điều kiện nào đó.

Cụ thể, cho $S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$ ký hiệu đặc tả của một toán tử *S*, ở đây S_{iv} là tập tất các biến đầu vào, giá trị của nó không được thay đổi bởi toán tử *S*, S_{ov} là tập tất cả các biến đầu ra, giá trị của nó được sinh ra hoặc được cập nhật bởi toán tử *S*, và S_{pre} và S_{post} tương ứng là precondition và postcondition. Đặc điểm của kiểu đặc tả này đó là postcondition S_{post} được sử dụng để miêu tả mối quan hệ giữa các trạng thái khởi tạo và các trạng thái cuối cùng. Chúng ta giả thiết rằng trong postcondition một biến như ˜x được sử dụng để biểu thị giá trị khởi tạo của biến *x* trước khi áp dụng toán tử, tức là *x* được sử dụng biểu diễn giá trị cuối cùng của *x* sau áp dụng toán tử. Do vậy, $\tilde{}x \in S_{iv}$ và $x \in S_{ov}$. Tất nhiên, S_{iv} cũng chứa tất cả các biến

đầu vào được khai báo như các tham số đầu vào và S_{ov} cũng gồm tất cả các biến đầu ra khác được khai báo như các tham số đầu ra.

Một chiến lược thực tế cho tạo ra các ca kiểm thử để thực thi các hành vi mong muốn của tất cả các kịch bản dòng dữ liệu được dẫn từ đặc tả được thiết lập dựa trên khái niệm của kịch bản dòng dữ liệu. Để miêu tả chính xác chiến lược này, đầu tiên chúng ta cần giới thiệu kịch bản dòng dữ liệu.

Định nghĩa 1:

Cho $S_{post} \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee \ldots \vee (C_n \wedge D_n)$, ở đây mỗi C_i $(i \in \{1,\ldots,n\})$ là một tiên đề, được gọi là *"guard condition",* không chứa biến đầu ra trong S_{ov}; D_i là một *"defining condition"* cái mà chứa ít nhất một biến đầu ra trong S_{ov}. Khi đó, một kịch bản dòng dữ liệu f_s của S là một sự kết nối $\tilde{S}_{pre} \wedge C_i \wedge D_i$ và biểu thức $((\tilde{S}_{pre} \wedge C_1 \wedge D_1) \vee (\tilde{S}_{pre} \wedge C_2 \wedge D_2) \vee \ldots \vee (\tilde{S}_{pre} \wedge C_n \wedge D_n))$ được gọi là kịch bản dòng dữ liệu của S.

Precondition $\tilde{S}_{pre} = S_{pre}(\tilde{\sigma}/\sigma)$ ký hiệu kết quả dự đoán từ việc thay thế trạng thái khởi tạo $\tilde{\sigma}$ thành trạng thái cuối cùng σ trong precondition S_{pre}. Chúng ta coi sự kết nối $\tilde{S}_{pre} \wedge C_i \wedge D_i$ như là một kịch bản dòng dữ liệu vì nó định nghĩa hành vi độc lập: khi $\tilde{S}_{pre} \wedge C_i$ được thỏa mãn bởi trạng thái khởi tạo, trạng thái cuối cùng (hoặc các biến đầu ra) được định nghĩa bởi defining condition D_i. Sự kết nối $\tilde{S}_{pre} \wedge C_i$ được biết như là *điều kiện kiểm thử* của kịch bản $\tilde{S}_{pre} \wedge C_i \wedge D_i$, cái này phục vụ như là hệ số cho việc tạo ra ca kiểm thử từ kịch bản dòng dữ liệu này.

Để áp dụng KỸ THUẬT DỰA VÀO KỊCH BẢN DÒNG DỮ LIỆU một cách hiệu quả, FSF của đặc tả phải thỏa mãn điều kiện *well-formed* được định nghĩa dưới đây.

Định nghĩa 2: Cho FSF của đặc tả S là $(\tilde{S}_{pre} \wedge C_1 \wedge D_1) \vee (\tilde{S}_{pre} \wedge C_2 \wedge D_2) \vee \ldots \vee (\tilde{S}_{pre} \wedge C_n \wedge D_n)$. Nếu S thỏa mãn điều kiện $(\forall_{i,j \in \{1,\ldots,n\}} \cdot (i \neq j \Rightarrow (C_i \wedge C_j \Leftrightarrow false))) \wedge (\tilde{S}_{pre} \Rightarrow (C_1 \vee C_2 \vee \ldots \vee C_n \Leftrightarrow true))$, S được nói là well-formed.

Well-formed của đặc tả S đảm bảo rằng mỗi kịch bản dòng dữ liệu định nghĩa một hàm độc lập và các guard condition bao hoàn toàn lĩnh vực được giới hạn một cách đầy đủ.

Với giả thiết rằng S là well-formed, chúng ta có thể tập trung vào sinh ca kiểm thử từ một kích bản chức năng đơn, $\tilde{S}_{pre \wedge C_i \wedge D_i}$, tại một thời điểm sử dụng phương pháp của chúng ta. Khi đó ca kiểm thử được sử dụng để chạy chương trình. Chúng ta hãy sử dụng toán tử *ChildFareDiscount.* Chức năng của ChildFareDiscount sử dụng ngôn ngữ đặc tả SOFL [15] dưới đây tương tự như VDM-SL cho các đặc tả toán tử.

Process *ChildFareDiscount(a: int, n_f: int) a_f: int*
Pre *a > 0 and n_f > 1*
Post *(a > 12 => a_f == n_f)*
and
*(a ≤ 12 => a_f == n_f – n_f * 0.5)*
End_process

Đặc tả trên miêu tả rằng đầu vào *a* (đại diện cho *age*) phải lớn hơn *0* và *n_f (normal_fare)* phải lớn hơn 1. Khi *a* lớn hơn 12, đầu ra *a_f (actual_fare)* sẽ bằng *n_f;* ngược lại, *a_f* sẽ giảm bớt 50% trên *n_f.*

Theo thuật toán được báo cáo trong nghiên cứu [4], có ba kịch bản dòng dữ liệu có thể được dẫn từ đặc tả này:

(1) a > 0 and n_f > 1 and a > 12 and a_f = n_f
*(2) a > 0 and n_f > 1 and a ≤ 12 and a_f = n_f – n_f*0.5*
(3) a ≤ 0 or n_f ≤ 1 and anything

Bảng 3.1: Ví dụ kiểm thử

Ca kiểm thử:	*a = 5, n_f = 2*
Điều kiện kiểm thử:	*a > 0 and n_f > 1 and a ≤ 12*
Kịch bản dòng dữ liệu:	*a > 0 and n_f > 1 and a ≤ 12 and a_f = n_f – n_f * 0.5*

Giả thiết đặc tả được viết lại theo chương trình dưới đây (giống như Java):

```
        int ChildFareDiscount (int a, int n_f) {
(1)     If (a > 0 && n_f > 1) {
(2)       if (a > 12)
(3)          a_f:= n_f;
(4)       else a_f:= n_f**2 – n_f – n_f*0.5;
(5)     return a_f;}
(6)     else System.out.println("precondition bị vi phạm");
(7)     }
```

Hiển nhiên, chúng ta có thể dẫn các đường dẫn dưới đây: [(1)(2)(3)(5)], [(1)(2)'(4)(5)], và [(1)'(6)]. Trong đường dẫn [(1)(2)'(4)(5)], (2)' nghĩa là thỏa hiệp của điều kiện $a > 12$ (tức là $a \leq 12$), và tương tự cách hiểu áp dụng tới (1)' trong đường dẫn [(1)'(6)]. Chúng ta cũng chèn thêm một số khuyết trong phép gán *a_f = n_f ** 2 – n_f – n_f * 0.5* (chính xác là *a_f = n_f – n_f * 0.5*), ở đây *n_f**2* nghĩa rằng *n_f* mũ 2 (tức là n_f^2).

Điểm yếu của phương pháp kiểm thử đó là nó chỉ có thể tìm thấy sự có mặt của các lỗi nhưng không tìm thấy sự vắng mặt của các lỗi. Ví dụ, chúng ta sinh ra một ca kiểm thử, *{(a, 5), (n_f, 2)}*, từ điều kiện kiểm thử *a > 0 and n_f > 1 and a* $\leq$ *12* của kịch bản dòng dữ liệu (2), được minh họa trong Bảng 3.1. Thực thi một chương trình với ca kiểm thử này, đường dẫn [(1)(2)'(4)(5)] sẽ được đi qua. Kết quả của thực thi là *a_f = 2**2 – 2 – 2*0.5 = 1*. Kết quả này không chỉ ra sự có sẵn của lỗi vì khi điều kiện kiểm thử *a > 0 and a_f > 1 and a* $\leq$ *12* được thỏa mãn bởi ca kiểm thử, defining condition *a_f = n_f – n_f * 0.5* cũng được thỏa mãn bởi đầu ra *a_f = 1* (vì *1 = 2 – 2*0.5* $\Rightarrow$ *true*), cái mà chứng minh mà trong *ca này*, chương trình thực hiện kịch bản dòng dữ liệu một cách chính xác. Nhưng đường dẫn lạ chứa một lỗi.

Một giải pháp cho vấn đề này thực thi một chứng minh dựa trên Logic Hoare để kiểm tra đường dẫn chính xác với kịch bản dòng dữ liệu. Chứng minh chính xác được mong muốn tự động một cách hoàn toàn để cho phép chúng ta tích hợp kỹ thuật này vào trong kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu. Để hiểu hơn, chúng ta cần giới thiệu vắn tắt tiên đề gán trong Logic Hoare.

3.3.2. Ký hiệu được sử dụng trong Logic Hoare

Cho *x:= E* là một phép gán: gán kết quả của đánh giá biểu thức *E* tới biến *x*. Tiên đề cho phép gán là:

$$\{Q(E/x)\}x := E\{Q\}$$

Biểu thức này thể hiện rằng phép gán *x:=E* là chính xác với post-assertion Q và được dẫn từ pre-assertion $Q(E/x)$, một kết quả dự đoán từ việc thay thế *E* cho tất cả các xảy ra của *x* trong *Q*. Post-assertion *Q* biểu diễn một điều kiện mà phải được thỏa mãn bởi biến *x* sau khi thực thi phép gán (phép gán có thể được coi như là toán tử cập nhật biến *x*). Để tạo ra post-condition Q là true sau khi thực thi, biểu thức *E* phải thỏa mãn *Q* trước khi thực thi, đó là, $Q(E/x)$ là true, vì *x* biểu diễn *E* sau khi thực thi.

3.3.3. Kỹ thuật kết hợp Logic Hoare với kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu - Phương pháp TBFV

Kỹ thuật kết hợp Logic Hoare với kỹ thuật kiểm thử dựa trên kịch bản được gọi là *kỹ thuật chứng minh hình thức dựa trên kiểm thử (Testing - Based Formal Verification, viết tắt là: TBFV)*. Kỹ thuật TBFV là một kỹ thuật được sử dụng để chứng minh sự chính xác của các đường dẫn chương trình được xác định bằng kỹ thuật dựa vào kịch bản dòng dữ liệu. Nguyên lý của kỹ thuật TBFV gồm có ba điểm sau:

- Sử dụng kỹ thuật dựa vào kịch bản dòng dữ liệu sinh các ca kiểm thử thích hợp để xác định *tất cả các đường dẫn xuất hiện (representative paths)* trong chương trình được kiểm thử; mỗi đường dẫn được sử dụng ít nhất một ca kiểm thử.
- Cho $\tilde{S}_{pre} \wedge C_i \wedge D_i$ *(i = 1, ..., n)* ký hiệu kịch bản chức năng và ca kiểm thử *t* được sinh ra từ điều kiện kiểm thử $\tilde{S}_{pre} \wedge C_i$. Cho $p = [sc_1, sc_2, \ldots, sc_m]$ là một đường dẫn chương trình, trong đó sc_j *(j = 1,..., m)* được gọi là *đoạn chương trình*, là một quyết định (tức là một dự đoán), một phép gán, một lệnh "return", hoặc một lệnh in. Giả thiết đường dẫn *p* được sử dụng trong ca kiểm thử *t*. Để chứng minh sự chính xác của *p* với kịch bản dòng dữ liệu, chúng ta hình thành một *bộ ba đường dẫn (path triple).*

$$\{\tilde{S}_{pre}\} p \{C_i \wedge D_i\}$$

Bộ ba đường dẫn này giống như như cấu trúc của bộ ba Hoare, nhưng nó được thay đổi thành đường dẫn đơn hơn là chương trình. Điều này có nghĩa rằng nếu precondition $\tilde{S}_{pre}$ của chương trình là true trước khi đường dẫn *p* được thực thi, postcondition $C_i \wedge D_i$ của đường dẫn *p* sẽ là true trên sự kết thúc của *p*.

- Áp dụng lặp đi lặp lại tiên đề phép gán (hoặc tiền đề) có thể dẫn một pre-assertion, được kí hiệu là p_{pre}, để hình thành biểu thức dưới đây:

$$\{\tilde{S}_{pre}(\tilde{x}/x)\} \{p_{pre}(\tilde{x}/x)\} p \{C_i \wedge D_i(\tilde{x}/x)\}$$

ở đây $\tilde{S}_{pre}(\tilde{x}/x)$, $p_{pre}(\tilde{x}/x)$ và $C_i \wedge D_i(\tilde{x}/x)$ tương ứng là kết quả dự đoán tương ứng từ việc thay thế mọi biến đầu vào $\tilde{x}$ tương ứng cho biến đầu vào *x* trong dự đoán. Các sự thay thế này là cần thiết để loại bỏ xung đột giữa các biến đầu vào và các biến được cập nhật bên trong.

Cuối cùng, nếu $\tilde{S}_{pre}(\tilde{x}/x) \Longrightarrow p_{pre}(\tilde{x}/x)$ được chứng minh có nghĩa rằng không có lỗi nào xuất hiện trên đường dẫn; ngược lại chỉ ra sự xuất hiện lỗi trên đường dẫn.

Chúng ta thấy rằng do ứng dụng của các tiên đề phép gán và phân đoạn không thay đổi chỉ gồm có thao tác bằng tay theo cú pháp, dẫn từ pre-assertion $p_{pre}(\tilde{x}/x)$ có thể được thực hiện một cách tự động, nhưng ẩn ý chứng minh một cách hình thức $\tilde{S}_{pre}(\tilde{x}/x) \Longrightarrow p_{pre}(\tilde{x}/x)$, chúng ta có thể viết đơn giản như sau $\tilde{S}_{pre} \Longrightarrow p_{pre}$ trong báo cáo này, không thể thực hiện một cách tự động, thậm chí với sự hỗ trợ của một bộ chứng minh lý thuyết như PVS, phụ thuộc vào độ phức tạp của $\tilde{S}_{pre}$ và p_{pre}. Nếu thu được một cách tự động đầy đủ theo ưu tiên cao nhất, chứng minh hình thức của ẩn ý này có thể được "thay thế" bởi một kiểm thử. Đó là, đầu tiên chúng ta sinh ra các giá trị mẫu cho các biến trong $\tilde{S}_{pre}$ và p_{pre}, và khi đó đánh giá chúng xem p_{pre} là false khi $\tilde{S}_{pre}$ là true. Nếu điều này là true, chúng ta nói rằng đường dẫn đang được chứng minh chứa một lỗi. Do kỹ thuật kiểm thử có sẵn trong các bài báo [6, 9], nên chúng tôi không cần trình lại chi tiết trong báo cáo này.

3.4. Áp dụng phương pháp TBFV

3.4.1. Áp dụng cho đoạn chương trình

Để thử nghiệm phương pháp TBFV, trong đoạn này chúng tôi sẽ trình bày một nghiên cứu áp dụng phương pháp TBFV để kiểm thử và chứng minh đoạn chương trình của *IC card system (hệ thống thẻ IC)* cho *JR commute train service (dịch vụ tàu điện chuyển mạch JR)* ở Tokyo. Qua thử nghiệm thấy rằng phương pháp TBFV về cơ bản sử dụng được và hiệu quả nhưng phương pháp này cũng đối diện vài thách thức hoặc vài giới hạn mà cần được giải quyết trong các nghiên cứu tiếp theo.

Hệ thống thẻ IC được thiết kế để cung cấp các dịch vụ chức năng sau: (1) Điều khiển truy cập đến và thoát từ một trạng thái đường ray, (2) Mua vé sử dụng thẻ IC, (3) Nạp thẻ IC bằng tiền mặt hoặc thông qua tài khoản ngân hàng, và (4) Mua vé đi lại trong khoảng thời gian (thời gian một tháng hoặc ba tháng). Do giới hạn về thời gian, tác giả không thể trình bày chi tiết tất cả, nhưng tác giả sẽ lấy một trong các hoạt động bên trong được sử dụng trong hệ thống thẻ IC, gọi là *ChildFareDiscount* đã được trình bày ở trên làm ví dụ để minh họa cách phương pháp TBFV được áp dụng. Chương trình *ChildFareDiscount* chứa ba đường dẫn, chúng ta cần chứng minh hình thức ba đường dẫn. Do vậy quá trình chứng minh cho ba

đường dẫn này là giống nhau nên tác giả chỉ cần chứng minh đường dẫn [(1)(2)'(4)(5)], đường dẫn này được sử dụng bởi ca kiểm thử *{(a, 5), (n_f,2)}*.

Đầu tiên chúng ta xây dựng bộ ba đường dẫn:

{˜a > 0 *and* ˜*n_f* > 1}
[a > 0 && *n_f* > 1
a ≤ 12,
a_f := *n_f* ∗∗2 − *n_f* − *n_f* ∗ 0.5,
return a_f]
{ ˜a ≤ 12 *and a_f* = ˜*n_f* − ˜*n_f* ∗ 0.5}

Ở đây ˜$a > 0$ *v*à ˜$n\_f > 1$ là kết quả thay thế ˜a *v*à ˜n_f cho các biến đầu vào a và n_f tương ứng trong pre-condition của chương trình, và ˜$a \leq 12$ *v*à $a\_f =$ ˜n_f − ˜n_f ∗ 0.5 là kết quả hoàn thành thay thế trong post-condition.

Thứ hai, chúng ta áp dụng lặp đi lặp lại tiên đề phép gán hoặc tiền đề phép gán với lệnh không thay đổi cho bộ ba đường dẫn [(1)(2)'(4)(5)], bắt đầu từ post-condition. Kết quả chúng ta xây dựng được đường dẫn dưới đây, được gọi là *asserted path (đường dẫn quyết định),* với các quyết định bên trong được dẫn từ hai đoạn chương trình*:*

{ ˜a > 0 *and* ˜*n_f* > 1}
{ ˜a ≤ 12 *and*
˜*n_f* ∗∗2 − ˜*n_f* − ˜*n_f* ∗ 0.5 = ˜*n_f* − ˜*n_f* ∗ 0.5}
a > 0 && *n_f* > 1
{ ˜a ≤ 12 *and*
n_f ∗ ∗2 − *n_f* − *n_f* ∗ 0.5 = ˜*n_f* − ˜*n_f* ∗ 0.5}

a ≤ 12
{˜a ≤ 12 *and*
n_f ∗ ∗2 − *n_f* − *n_f* ∗ 0.5 = ˜*n_f* − ˜*n_f* ∗ 0.5}
a_f := *n_f* ∗∗2 − *n_f* − *n_f* ∗ 0.5
{ ˜a ≤ 12 *and a f* = ˜*n_f* − ˜*n_f* ∗ 0.5}
return a_f
{˜a ≤ 12 *and a_f* = ˜*n_f* − ˜*n_f* ∗ 0.5}

Ở đây đường dẫn quyết định ˜$a \leq 12$ *v*à ˜n_f ∗∗ 2 − ˜n_f − ˜n_f ∗ 0.5 = ˜n_f − ˜n_f ∗ 0.5, dòng thứ hai từ trên xuống, là kết quả thay thế ˜a cho a và ˜n_f cho n_f trong quyết định { ˜$a \leq$

$12\ v\grave{a}\ n\_f \quad ** 2 - n\_f - n\_f * 0.5 = \ \tilde{}n\_f - \tilde{}n\_f * 0.5\}$. Như đã trình bày ở trên, điều này là cần thiết để giữa sự tin cậy của các biến đầu vào *a* và *n_f* trong pre-condition gốc (biểu thị là $\tilde{}a\ v\grave{a}\ \tilde{}n\_f$) và pre-assertion.

Thứ ba, chúng ta cần đánh giá tính hợp lý của ẩn ý $\tilde{}a > 0\ v\grave{a}\ \tilde{}n\_f > 1 \Rightarrow \tilde{}a \leq 12\ v\grave{a}\ \tilde{}n\_f \quad ** 2 - \ \tilde{}n\_f \quad - \tilde{}n\_f \quad * 0.5 = \ \tilde{}n\_f \quad - \tilde{}n\_f \quad * 0.5$. Sử dụng ca kiểm thử $\{(\tilde{}a, 5), (\tilde{}n\_f \quad , 8)\}$, chúng ta có thể dễ dàng chứng minh rằng ẩn ý này là sai (đánh giá chi tiết không được đề cập vì giới hạn thời gian).

Từ ví dụ trên, chúng ta có thể thấy rằng đôi khi kiểm thử thậm chí có thể hiệu quả hơn chứng minh hình thức trong việc đánh giá tính hợp lý của ẩn ý khi một lỗi có sẵn trên đường dẫn, nhưng nếu đường dẫn không chứa lỗi, về cơ bản kiểm thử sẽ không thể phát hiện để đưa ra một kết luận. Trong trường hợp này, một đánh giá kỹ thuật phải được tạo ra cho việc đánh giá tính hợp lý. Điểm mạnh của kiểm thử đó là có thể thực hiện được tự động, đây là điều vô cùng có ích trong thời đại công nghiệp.

3.4.2. Áp dụng cho việc gọi phương thức

Nếu một gọi phương thức (method invocation) được sử dụng như là câu lệnh, chúng ta có thể thay đổi trạng thái hiện tại của chương trình ChildFareDiscount. Do vậy, đường dẫn bên trong phương thức được gọi sẽ phải được xem xét trong pre-assertion của chương trình dưới dạng kiểm thử.

Chúng ta hãy thay đổi chương trình *ChildFareDiscount* và tổ chức hoàn thiện chương trình này thành một lớp (class) được gọi là *FareDiscount* dưới đây.

```
        class  FareDiscount{
        int tem; //instance variable

        int ChildFareDiscount1(int a, int n_f){
(1)       Discount(n_f);
(2)       if (a > 0 && n_f  > 1){
(3)       if (a > 12)
(4)           a_f := n_f
(5)       else a_f := n_f **2 – n_f  – tem;
(6)       return a_f; }
(7)       else System.out.println("the precondition is violated");
        }
```

```
        void Discount(int x){
        int r;
(1.1)    r :=x*0.5;
(1.2)    tem := r; }
      }
```

Khi chạy phương thức *ChildFareDiscount1* trong đó phương thức *Discount(n_f)* được gọi, chúng ta lấy được ba đường dẫn: [(1)(2)(3)(4)(6)], [(1)(2)(3)'(5)(6)] và [(1)(2)'(7)], ở đây đoạn (1) là một đường dẫn con (subpath) [(1.1)(1.2)](n_f/x), biểu thị đường dẫn kết quả từ việc thay thế tham số thực tế *n_f* cho tham số hình thức *x* trong đường dẫn con [(1.1)(1.2)]. Do vậy đường dẫn [(1)(2)(3)'(5)(6)] thực tế sau khi chèn thêm đường dẫn trong *Discount* vào đường dẫn trong ChildFareDiscount1 được biểu diễn như sau [(1.1)(1.2)(2)(3)'(5)(6)]. Lựa chọn ca kiểm thử giống nhau {(a,5), (n_f,2)} trước khi chạy chương trình, chúng ta tạo ra đường dẫn [(1.1)(1.2)(2)(3)'(5)(6). Khi đó chúng ta xây dựng được đường dẫn quyết định (asserted path) như sau:

$\{\tilde{a} > 0 \ \ and \ \ \tilde{n\_f} > 1\}$
$\{\tilde{a} \leq 12 \ and$
$\tilde{n\_f} * *2 - \tilde{n\_f} - \tilde{n\_f} * 0.5 = \tilde{n\_f} - \tilde{n\_f} * 0.5\}$
$r := n\_f * 0.5$
$\{\tilde{a} \leq 12$
$n\_f * *2 - n\_f - r = \tilde{n\_f} - \tilde{n\_f} * 0.5\}$
$tem := r$
$\{\tilde{a} \leq 12 \ and$
$n\_f * *2 - n\_f - tem = \tilde{n\_f} - \tilde{n\_f} * 0.5\}$
$a > 0 \ \&\& \ n\_f > 1$
$\{\tilde{a} \leq 12 \ and$
$n\_f * *2 - n\_f - tem = \tilde{n\_f} - \tilde{n\_f} * 0.5\}$
$a \leq 12$
$\{\tilde{a} \leq 12 \ and$
$n\_f * *2 - n\_f - tem = \tilde{n\_f} - \tilde{n\_f} * 0.5\}$
$a\_f := n\_f * *2 - n\_f - tem$
$\{\tilde{a} \leq 12 \ and \ a\_f = \tilde{n\_f} - \tilde{n\_f} * 0.5\}$
$return \ \ a\_f$
$\{\tilde{a} \leq 12 \ and \ a\_f = \tilde{n\_f} - \tilde{n\_f} * 0.5\}$

Ở đây đường dẫn con *[r:=n_f*0.5, tem:= r]* là kết quả thay thế tham số thực tế *n_f* được sử dụng trong lời gọi phương thức *Discount(n_f)* cho tham số hình thức *x* được sử dụng trong định nghĩa phương thức trong đường dẫn con gốc *[r:=x*0.5, tem:= r]*. Tương tự, chúng ta có thể dễ dàng sử dụng kiểm thử để chứng minh ẩn ý $\tilde{}a > 0$ và $\tilde{}n\_f > 1 \Rightarrow \tilde{}a \leq 12$ và $\tilde{}n\_f \ ** 2 - \tilde{}n\_f \ - \tilde{}n\_f \ * 0.5 = \tilde{}n\_f \ - \tilde{}n\_f \ * 0.5$ là sai, chỉ ra rằng một lỗi được tìm thấy trên đường dẫn.

3.4.3. Các nghiên cứu liên quan

Một trong những kết quả đạt được là trình bày Design By Contract (DBC) của Meyer đã được ứng dụng trong ngôn ngữ lập trình Eiffel [7, 8]. Thành công của Eiffel đó là kiểm tra pre-condition và post-condition và khuyến khích môn học DBC trong lập trình để phát triển nghiên cứu tương tự cho các ngôn ngữ khác như hệ thống kiểm thử Sunit cho Smalltalk [19]. Cheon và Leavens miêu tả một phương pháp kiểm thử đơn vị (unit testing) mà sử dụng một bộ kiểm tra assertion thời gian chạy của ngôn ngữ đặc tả hình thức để quyết định các phương thức làm việc chính xác theo đặc tả hình thức sử dụng pre-condition và post-condition, và đã cài đặt thành công ý tưởng này sử dụng ngôn ngữ mô hình Java (Java Modeling Language – JML) và nền tảng làm việc kiểm thử Junit [12]. Gray và Mycroft miêu tả phương pháp khác để kiểm thử các chương trình Java sử dụng các đặc tả kiểu Hoare [18].

3.5. Kết luận

Nguyên tác cơ bản của TBFV đầu tiên sử dụng kiểm thử dựa trên kịch bản dòng dữ liệu để đưa ra một đường dẫn của chương trình dưới hình thức kiểm thử, và khi đó áp dụng phương pháp dựa Logic Hoare để chức minh hình thức sự chính xác của mỗi đường dẫn. Phương pháp TBFV có một ưu điểm so với chứng minh sự chính xác hình thức dựa trên Logic Hoare đó là có thể thực hiện tự động bằng cách xây dựng hệ thống chương trình thực tế. Phương pháp này cũng có ưu điểm nổi bật trong việc giảm số lượng các ca kiểm thử cần thiết so với kiểm thử dựa trên đặc tả có sẵn.

KẾT LUẬN VÀ KIẾN NGHỊ

1. Kết luận

Về mặt cơ bản em đã hoàn thành được mục tiêu của đề tài đưa ra. Một số kết quả đạt được như sau:

- Nắm được kiến thức cơ bản liên quan đến Kiểm thử phần mềm và kiểm thử tĩnh;

- Nắm được các kỹ thuật kiểm thử tĩnh và các phương pháp kiểm thử dòng dữ liệu tĩnh trong kiểm thử phần mềm;

- Hiểu được Logic Hoare trong việc chứng minh sự chính xác của chương trình và nghiên cứu được kỹ thuật kết hợp Logic Hoare với kỹ thuật kiểm thử dựa vào kịch bản dòng dữ liệu để nâng cao hiệu quả cho kỹ thuật kiểm thử dựa vào kịch bản dòng dữ liệu.

- Quyển báo cáo có thể làm tài liệu tham khảo về lĩnh vực Kiểm thử phần mềm, Kiểm thử tĩnh và đặc biệt là kiểm thử dòng dữ liệu tĩnh;

- Kết quả nghiên cứu có thể làm tiền đề cho các nghiên cứu liên quan khác.

2. Kiến nghị

Với thời gian nghiên cứu ngắn và đây là lĩnh vực mới tiếp cận nên trong báo cáo còn một số phần chưa được hoàn thiện. Do vậy thời gian tới em sẽ tiếp tục nghiên cứu chuyên sâu hơn và cố gắng xây dựng được chương trình kiểm thử tự động dựa vào kỹ thuật kết hợp giữa Logic Hoare với kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu.

Trong quá trình làm luận văn, em đã cố gắng rất nhiều, tuy nhiên không tránh khỏi những thiếu sót, em mong rằng sẽ nhận được các ý kiến đóng góp của các thầy cô, các bạn bè, đồng nghiệp để bản luận văn ngày càng hoàn thiện hơn.

TÀI LIỆU THAM KHẢO

Tiếng Việt:

[1] Phạm Ngọc Hùng, Trương Anh Hoàng và Đặng Văn Hưng (2014), *Giáo trình kiểm thử phần mềm.*

Tiếng Anh:

[2] Bath, G., McKay, J.: Praxiswissen Softwaretest - Test Analystund Technical Test Analyst. Dpunkt, Heidelberg (2010).

[3] Beck (2002), *Test driven development: By example*, Addison Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[4] Fosdick Lloyd D. and Osterweil Leon J. (1976), *Data flow analysis in software reliability*, ACM Comput. Surv. 8, no. 3, 305–330.

[5] Huang J. C. (1979), *Detection of data flow anomaly through program instrumentation*, IEEE Trans. Softw. Eng. 5, no. 3, 226–236

[6] K. B. Gallagher and J. R. Lyle, "Using program slicing in software mainte-nance.," IEEE Trans. Software Eng. , vol. 17, no. 8, pp. 751–761, 1991.

[7] Liggesmeyer (2009), P.: Software-Qualität: Testen, Analysieren und Ve rifizieren von Softwa re, 2nd edn. Spektrum-Akademischer Verlag, Berlin.

[8] M. Weiser, *"Program slicing.,"* IEEE Trans. Software Eng., vol. 10, no. 4, pp. 352–357, 1984.

[9] M. Weiser, *"Program slicing.,"* in IC SE , pp. 439–449, 1981.

[10] Majchrzak, T.A., Kuchen, H.: IHK-Projekt Softwaretests: Auswertung. In: Working Pa pers, Vol. 2. Förderkreis der Angewa ndten Informatik an der Westfälischen Wilhelms-Universität, Münster e .V. (2010).

[11]Michael Kart (2012), *Behavior-driven development: conference tutorial*, J. Comput. Sci.Coll. 27, no. 4, 75–75.

[12] P. C. Jorgensen, *Software Testing: A Craftsman's Approach* . CRC Press, 2nd ed., 2002.

[13] Pezze, M., Young, M.: *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, New York (2007).

[14] Roitzsch, E .H.P.: Analytische Softwarequalitätssicherung in Theorie und Praxis: Der Wegzur Software mit hoher Qualität durch statisches Prüfen, dynamisches Testen, formales Beweisen. Monsensteinund Vannerdat (2005).

[15] S.Liu (2004), *Formal Engineering for Industrial Software Development Using the SOFL Method*. Springer-Verlag, ISBN 3-540-20602-7.

[16] S. Liu and S. Nakajima (2010), *A Decompositional Approach to Automatic Test Case Generation Based on Formal Specifications*. In 4th IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2010), pages 147 {155, Singapore, June 9-11 2010. IEEE CS Press}.

[17] S. Liu, T. Hayashi, K. Takahashi, K. Kimura, T. Nakayama, and S. Nakajima (2010), *Automatic Trans-formation from Formal Specifications to Functional Scenario Forms for Automatic Test Case Genneration*. In 9th International Conference on Software Methodologies, Tools and Techniques (SoMet 2010), page to appear, Yokohama city, Japan, Sept. 29- Oct. 1 2010. IOS International Publisher.

[18] S. Rapps and E. J. Weyuker, "*Selecting software test data using data flow information.,"* IEEE Trans. Software Eng. , vol. 11, no. 4, pp. 367–375, 1985.

[19] S.Liu and S.Nakajima (2011), *A "Vibration" method for Automatically Generating Test Cases Based on Formal Specifications*. In 18th Asia-Pacific Software Engineering Conference (APSEC 2011), pages 73{80, HCM City, Vietnam, Dec. 5-8 2011. IEEE CS Press.

[20] Shaoying Liu, *"Utilizing Hoare Logic to Strengthen Testing for Error Detection in Programs"*.

[21] Sneed, H.M., Winter, M.: *Testen Objektorientierter Software*. Hanser, München (2002).

[22] Zeller, A.: Why Programs Fail: *A Guide to Systematic Debugging*. Morgan Kaufmann, San Francisco (2006).